Linux Exploit Mitigation

Dobin Rutishauser
V1.3, March 2016
About me

At Compass Security since 2011

Spoke at OWASP Zürich, Bsides Vienna

On the internet:

- @dobinrutis
- github.com/dobin
To understand exploit mitigations

Need to understand exploit techniques

I’ll lead you all the way, from zero

In 45 minutes!
About this presentation

- Content of 8 hours for BFH
- It will get very technical
- Not possible to:
  - Cover all the topics
  - And be easy to understand
  - And handle all the details
- This should give more of an … overview
- Don’t worry if you don’t understand everything
Overview of the presentation
Overview

1. Memory Layout
2. Stack
3. Exploit Basics
4. Exploit Mitigation
   • DEP
   • Stack Protector
   • ASLR
5. Contemporary Exploiting
6. Hardening
7. Container
8. Kernel
Exploit Intention

Attacker wants:

- Execute his own code on the server
  - `rm -rf /`
- Connect-back shellcode
- `echo "sysadmin:::" >> /etc/passwd`
Exploit Requirements

Attacker needs:
- Be able to upload code to execute
- Be able to hijack instruction flow
Memory Corruption Types

Memory Corruptions

Buffer Overflow

- `strncpy / strcat / memcpy`
- Write past allocated buffer
- Minimum: 1 byte (off by one)

Alternative: Arbitrary Write

- Can write certain memory area
- Not in scope in this presentation
History intermezzo

Morris Worm in 1988, overflow in sendmail and finger

Around year 2000: Golden age of remote exploits

Team Teso (formatstring vulnerabilities)
  ✦ PHP, Apache, telnetd, wu-ftpd, qpopper, …

w00w00 (heap overflows)
  ✦ Efnet ircd, norton antivirus, AOL messenger, unixware stuff

Gobbles (putting the lulz in)
  ✦ Apache Scalp

ADM (high quality exploits)
  ✦ Bind, wu-imapd, …
Userspace Memory Layout
Process Memory Layout

Stack

Heap

Code

0xc0000000
0xbfffffff

0x0804800
0x00000000
Process Memory Layout

Stack

Heap

Code

Program

char array[16];
malloc(16)

mapping
Userspace process data structures

Stack
- There’s one contiguous memory region containing the stack for the process
- LIFO – Last In, First Out
- Contains function local variables
- Also contains: Saved Instruction Pointer (SIP)
- Current function adds data to the top (bottom) of the stack

Heap
- There’s one contiguous memory region containing the heap
- Memory allocator returns specific pieces of the memory region
- For malloc()
- Also contains: heap management data

Code
- Compiled program code
Stacks

How do they work?
Stack

push

pop
Stack

push

pop

0x00010
0x10000
void main(void) {
    int blubb = 0;
    foobar(blubb);
    return;
}

void foobar (int arg1) {
    char compass1[];
    char compass2[];
}
Stack Layout

Saved IP (&__libc_start)
Saved Frame Pointer
Local Variables <main>

SIP
SFP
blubb

Stack Frame <main>

Argument arg1 for <foobar>

Saved IP (&return)
Saved Frame Pointer
Local Variable 1
Local Variable 2

&blubb
SIP
SFP
compass1
compass2

Stack Frame <foobar>

push
pop
Stack Layout

Saved IP (&__libc_start)
Saved Frame Pointer
Local Variables <main>

Argument arg1 for <foobar>
Saved IP (&return)
Saved Frame Pointer
Local Variable 1
Local Variable 2

Stack Frame <main>

Stack Frame <foobar>

push
pop
Stack

```c
void main(void) {
    int blubb = 0;
    foobar(blubb);
    return;
}

void foobar (int arg1) {
    char compass1[];
    char compass2[];
}
```
Stack Layout

SIP: Stored Instruction Pointer

- Copy of EIP
- Points to the address where control flow continues after end of function
  - (return, ret)
- Usually points into the code section
Stack Layout

Writers go up

Stack grows down

0xFFFF

arg1
SIP
SFP
localvar1
localvar2

push
pop
Recap! Memory Layout

User data is on the stack

Also: important stuff is on the stack (Instruction Pointer, SIP)

Stack grows down ↓

Writes go up ↑
Stack Overflow Exploitation
Exploitation Basics

• Program execution HIGHLY predictable/deterministic
  • Which is kind of surprising

• Stack, Heap, Code all start at the same address

• Same functions gets called in the same order
  • And allocate the same sized buffers

• “Error/Overflow in function X”, every time:
  • Same call stack
  • Same variables
  • Same registers
Buffer Overflow Basic Layout

char `buf1`[16]

EIP

0x0100

0xFFFF
Buffer Overflow Basic Layout

```c
char buf1[16] EIP

strcpy(buf1, "AAAA AAAAA AAAAA AAAAA ");

AAAA AAAAA AAAAA AAAAA FF12

(0xFF12 = address of previous function)
```

Write up
Buffer Overflow Basic Layout

```c
char buf1[16]

strcpy(buf1, "AAAA AAAAA AAAAA AAAAA BBBBB");
```

Attacker can call any code he wants
But: What code?
Buffer Overflow Basic Layout

Problem: In-band signaling
- Control data
- User data

Like old telephone networks
- 2600 hz: Indicate line is free
- With a 2600hz tone, you could phone anywhere, for free
- Oups, accidently created Legion of Doom
Buffer Overflow Basic Exploit

Return to Stack:

```
char buf1[16]  EIP
AAAAA AAAAA AAAAA AAAAA  BBBB

CODE CODE CODE CODE CODE CODE CODE &buf1
```

Jump to buffer with shellcode
Buffer Overflow Basic Exploit

Jump to buffer with shellcode
How is shellcode formed?

Short description of shellcode
Shellcode!

```
08048060 <_start>:
  8048060: 31 c0    xor %eax,%eax
  8048062: 50    push %eax
  8048063: 68 2f 2f 73 68    push $0x68732f2f
  8048068: 68 2f 62 69 6e    push $0x6e69662f
  804806d: 89 e3    mov %esp,%ebx
  804806f: 89 c1    mov %eax,%ecx
  8048071: 89 c2    mov %eax,%edx
  8048073: b0 0b    mov $0xb,%al
  8048075: cd 80    int $0x80
  8048077: 31 c0    xor %eax,%eax
  8048079: 40    inc %eax
  804807a: cd 80    int $0x80

char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
  "\x68\x68\x2f\x62\x69\x66e\x89"
  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
  "\xcd\x80\x31\xc0\x40\xcd\x80";
```
Recap! Stack Overflow Exploit

Write past buffer on stack

Overwrite sIP

Point sIP to beginning of buffer

Place shellcode in buffer

Shellcode will be executed!
Exploit Mitigations
Exploit Mitigations

- DEP
- Stack Canary
- ASLR
Exploit Mitigation: DEP
DEP

DEP – Data Execution Prevention

- Aka: No-Exec Stack
- Aka: W^X (Write XOR eXecute)(OpenBSD)
- Aka: NX (Non-Execute) Bit

AMD64 (x86-64) introduced NX bit in HW

- Intel 32 bit architecture (starting from 80386) “saved” Xecute bit
- For 32 bit, need PAE (Physical Address Extension, 32->36bit)
- Or kernel patches like PaX

Linux:
- Support in 2004, Kernel 2.6.8, default
Anti-Exploitation: No-Exec Stack

Permissions: rwx

```
CODE CODE CODE CODE CODE CODE CODE &buf1
```

```
jmp *buf1
```
Anti-Exploitation: No-Exec Stack

Permissions: rw-

```
CODE CODE CODE CODE CODE CODE CODE CODE &buf1
```

```
jmp *buf1
```

“Segmentation Fault”
Read Only Stack

Program Headers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
<th>FileSiz</th>
<th>MemSiz</th>
<th>Flg</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHDR</td>
<td>0x000034</td>
<td>0x08048034</td>
<td>0x08048034</td>
<td>0x00120</td>
<td>0x00120</td>
<td>R E</td>
<td>0x4</td>
</tr>
<tr>
<td>INTERP</td>
<td>0x000154</td>
<td>0x08048154</td>
<td>0x08048154</td>
<td>0x00013</td>
<td>0x00013</td>
<td>R</td>
<td>0x1</td>
</tr>
<tr>
<td>GNU_FRAME</td>
<td>0x0004d0</td>
<td>0x080484d0</td>
<td>0x080484d0</td>
<td>0x00034</td>
<td>0x00034</td>
<td>R</td>
<td>0x4</td>
</tr>
<tr>
<td>GNU_STACK</td>
<td>0x000000</td>
<td>0x00000000</td>
<td>0x00000000</td>
<td>0x00000</td>
<td>0x00000</td>
<td>RW</td>
<td>0x4</td>
</tr>
<tr>
<td>GNU_RELRO</td>
<td>0x000f14</td>
<td>0x08049f14</td>
<td>0x08049f14</td>
<td>0x000ce</td>
<td>0x000ce</td>
<td>R</td>
<td>0x1</td>
</tr>
</tbody>
</table>
DEP: Memory Layout

Stack

Heap

Code

0x00000000

EIP
```c
compass@ubuntu:~/bof$ cat shellcode.c

int main(void)
{
    char shellcode[] =
    "\x48\x31\xd2" // xor %rdx, %rdx
    "\x48\xbb\x2f\x62\x69\xe2\x73\x68" // mov $0x68732f6e69622f2f, %
    rbx
    "\x48\xc1\xe8\x08" // shr $0x8, %rbx
    "\x53" // push %rbx
    "\x48\x89\xe7" // mov %rsp, %rdi
    "\x50" // push %rax
    "\x57" // push %rdi
    "\x48\x89\xe6" // mov %rsp, %rsi
    "\xb0\x3b" // mov $0x3b, %al
    "\x0f\x05";

    (*(void (*)()) shellcode)();

    return 0;
}
compass@ubuntu:~/bof$ gcc shellcode.c -o sh && ./sh
Segmentation fault (core dumped)
compass@ubuntu:~/bof$ gcc -z execstack shellcode.c -o sh-execstack && ./sh-execstack
$ id
uid=1000(compass) gid=1000(compass) groups=1000(compass),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare)
$
Recap! DEP

Exploit Mitigation – DEP

- Makes it impossible for an attacker to execute his own shellcode
- Code: eXecute (no write)
- Heap, Stack: Write (no execute)

- No-no: Write and Execute
  - Sometimes necessary
  - Interpreted Languages
  - E.g. Java
  - Or JavaScript
  - Ähem *Browser* ähem
Exploit Mitigation – Stack Protector
Exploit Mitigation – Stack Protector

• Aka:
  • SSP: Stack Smashing Protector
  • Stack Cookie
  • Stack Canary

• Secret value in front of control data

• Generated per-process
  • Not per-function
Exploit Mitigation – Stack Protector

```
char buf1[16]    EIP
```
Exploit Mitigation – Stack Protector

<table>
<thead>
<tr>
<th>char buf1[16]</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>char buf1[16]</td>
<td>secret</td>
</tr>
</tbody>
</table>
Exploit Mitigation – Stack Protector

<table>
<thead>
<tr>
<th>char buf1[16]</th>
<th>secret</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>char buf1[16]</td>
<td>55667</td>
<td>FF12</td>
</tr>
</tbody>
</table>

CODE CODE CODE CODE CODE

BBBB AA00
Exploit Mitigation – Stack Protector

```
char buf1[16]  55667  FF12

CODE CODE CODE CODE CODE BBBB AA00
```

“Segmentation Fault”  BBBB != 55667
Exploit Mitigation – Stack Protector

Stack Protector

- GCC patch
  - First: StackGuard in 1997
  - Then: ProPolice in 2001, by IBM
  - Finally: Re-implement ProPolice in 2005 by RedHat
  - introduced in GCC 4.1
  - -fstack-protector

- Update: Better implementation by Google in 2012
  - -fstack-protector-strong

- Enabled since like forever by default
  - most distributions
  - most packages
Exploit Mitigation – Stack Protector

DEADPOOL APPROVES

DO YOU?
Exploit Mitigation: ASLR
Exploit Mitigation - ASLR

• Code execution is surprisingly deterministic

• E.g. Network service:
  1. fork()
  2. Parse incoming data
  3. Buffer Overflow is happening at module X line Y

• On every exploit attempt, memory layout looks the same!
  • Same stack/heap/code layout
  • Same address of the buffer(s)

• ASLR: Address Space Layout Randomization
  • Introduces randomness in memory regions
Memory Layout

0xbfffffff
Stack

Heap

Code

0x0804800
Exploit Mitigation - ASLR

0xAA00

CODE CODE CODE CODE CODE C &buf1

0xAA00

CODE CODE CODE CODE CODE C AA00
Exploit Mitigation - ASLR

```
0xBB00

CODE CODE CODE CODE CODE C &buf1

0xBB00

CODE CODE CODE CODE CODE C AA00
```

“Segmentation Fault”

```
AA00 != BB00
```
Exploit Mitigation - ASLR

Randomness is measured in entropy

- Several restrictions
  - Pages have to be page aligned: 4096 bytes = 12 bit
  - Very restricted address space in x32 architecture
    - ~8 bit for stack (256 possibilities)
  - Much more space for x64
    - ~22 bit for stack

Re-randomization

- ASLR only applied on exec()
  - With some bugs…
- Not on fork()
Recap! ASLR

Randomize Memory Layout

Attacker can’t call/reference ✦ what he can’t find

Default ASLR randomizes:
✦ Writeable locations
✦ Stack
✦ Heap
Recap! All Exploit Mitigations

Stack canary: detects/blocks overflows

DEP: makes it impossible to execute uploaded code

ASLR: makes it impossible to locate data
Stack canary, DEP, ASLR… so many protections… now I’m secure!

Time for beer and pizza!
This was the simple part…
Contemporary exploiting
Contemporary exploiting

Defeating: Stack Canary
Exploiting: Stack Canary

• Stack canary protects only **overflows**

• Arbitrary write!

```c
char array[16];
array[userIndex] = 0;
```

• Also: Heap is not protected

• Also: Local Vars (function ptr) not protected
Exploiting: Stack Canary

Or… just bruteforce it!

- 32 bit value, so $2^{32} \approx 4$ billion possibilities?
- Example: 0x42A1B2C3

<table>
<thead>
<tr>
<th>AAAAAAAAA</th>
<th>0x42</th>
<th>0xA1</th>
<th>0xB2</th>
<th>0xC3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAAAA</td>
<td>0x41</td>
<td>0xA1</td>
<td>0xB2</td>
<td>0xC3</td>
</tr>
<tr>
<td>AAAAAAAAAA</td>
<td>0x42</td>
<td>0xA1</td>
<td>0xB2</td>
<td>0xC3</td>
</tr>
<tr>
<td>AAAAAAAAAA</td>
<td>0x42</td>
<td>0x41</td>
<td>0xB2</td>
<td>0xC3</td>
</tr>
<tr>
<td>AAAAAAAAAA</td>
<td>0x42</td>
<td>0x42</td>
<td>0xB2</td>
<td>0xC3</td>
</tr>
</tbody>
</table>

A -> Crash
B -> No crash
BA -> crash
BB -> crash
Exploiting: Stack Canary

• So: not $2^{32} = 4$ billion possibilities

• But: $4 \times 8 = 4 \times 256 = 1024$ possibilities
  • 512 on average
Exploiting: Stack Canary

I lied a bit!

Argument for `<foobar>`
Saved IP (&main)
Saved Frame Pointer
Local Variables `<func>`

Stack Frame `<foobar>`

- `arg1`
- `SIP`
- `SFP`
- `compass1`
- `compass2`

push  pop
Recap: Defeating Stack Canary

- Defeat ASLR for free, because brute force sFP 😊
- Conclusion: Stack Canary is can be brute forced, or just circumvented
Contemporary exploiting

Defeating: DEP
Exploiting: DEP - Memory Layout

- Stack: rw-
- Heap: rw-
- Code: r-x

0x0804800
Exploiting: DEP - ROP

• DEP does not allow execution of our own code

• But what about existing code?

• Code from binary, followed by a RET
  • Called “gadgets”

• Return Oriented Programming
Exploiting DEP: ROP Gadgets

ROPgadget

```
x00000000000440608 : mov dword ptr [rdx], ecx ; ret
x000000000004598b7 : mov eax, dword ptr [rax + 0xc] ; ret
x00000000000431544 : mov eax, dword ptr [rax + 4] ; ret
x0000000000045a295 : mov eax, dword ptr [rax + 8] ; ret
x000000000004a3788 : mov eax, dword ptr [rax + rdi*8] ; ret
x00000000000493dec : mov eax, dword ptr [rdx + 8] ; ret
x000000000004a36f7 : mov eax, dword ptr [rdx + rax*8] ; ret
x00000000000493dc8 : mov eax, dword ptr [rsi + 8] ; ret
x0000000000043fbeb : mov eax, ebp ; pop rbp ; ret
x000000000004220fa : mov eax, ebx ; pop rbx ; ret
x00000000000495b90 : mov eax, ecx ; pop rbx ; ret
x00000000000482498 : mov eax, edi ; pop rbx ; ret
x00000000000437c11 : mov eax, edi ; ret
x0000000000042cfa1 : mov eax, edx ; pop rbx ; ret
x0000000000047d484 : mov eax, edx ; ret
x0000000000043de7e : mov ebp, esi ; jmp rax
x00000000000499461 : mov ecx, esp ; jmp rax
x000000000004324fb : mov edi, dword ptr [rbp] ; call rbx
x00000000000443f34 : mov edi, dword ptr [rdi + 0x30] ; call rax
x000000000004607e2 : mov edi, dword ptr [rdi] ; call rsi
x0000000000045c71e : mov edi, ebp ; call rax
x00000000000491e33 : mov edi, ebp ; call rdx
x000000000004a7a2d : mov edi, ebp ; nop ; call rax
x0000000000045c4c1 : mov edi, ebx ; call rax
```
ROPgadet

ROPgadget.py --ropchain

```
ROP chain generation

Step 1 -- Write-what-where gadgets

+] Gadget found: 0x806f702 mov dword ptr [edx], ecx ; ret
+] Gadget found: 0x8056c2c pop edx ; ret
+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
[-] Can't find the 'xor ecx, ecx' gadget. Try with another 'mov [r], r'
+] Gadget found: 0x808fe0d mov dword ptr [edx], eax ; ret
+] Gadget found: 0x8056c2c pop edx ; ret
+] Gadget found: 0x80c5126 pop eax ; ret
+] Gadget found: 0x80488b2 xor eax, eax ; ret

Step 2 -- Init syscall number gadgets

+] Gadget found: 0x80488b2 xor eax, eax ; ret
+] Gadget found: 0x807030c inc eax ; ret

Step 3 -- Init syscall arguments gadgets

+] Gadget found: 0x80481dd pop ebx ; ret
+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
+] Gadget found: 0x8056c2c pop edx ; ret

Step 4 -- Syscall gadget

+] Gadget found: 0x804936d int 0x80

Step 5 -- Build the ROP chain

#!/usr/bin/env python2
# execve generated by ROPgadget v5.2

from struct import pack

# Padding goes here
p = ''
p += pack('<I', 0x8056c2c) # pop edx ; ret
p += pack('<I', 0x80f4060) # @ .data
p += pack('<I', 0x80c5126) # pop eax ; ret
p += '/bin/

p += pack('<I', 0x808fe0d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x8056c2c) # pop edx ; ret
p += pack('<I', 0x80f4064) # @ .data + 4
p += pack('<I', 0x80c5126) # pop eax ; ret
p += '/sh'
```
Exploiting: DEP - Memory Layout

Stack

Heap

Code

0x0804800
Exploiting: ROP

Stager:

• Allocate new RWX memory
• Copy rest of shellcode to newly allocated memory
• Execute it (jmp)
• Profit
Recap: Anti-DEP

Conclusion:

Code section is not randomized

Just smartly re-use existing code
Contemporary exploiting

Defeating: ASLR
Exploiting: ASLR

I lied… again
Exploiting: ASLR – Memory Layout

- Stack
  - Address: 0x???

- Heap
  - Address: 0x???

- Code
  - Address: 0x0804800
Exploiting: ASLR – Memory Layout

Stack

Mappings

Heap

Code

Libc
ssl
libstdc++

0x0804800

0x????????

© Compass Security Schweiz AG
www.csnc.ch
Exploiting: ASLR – Memory Layout

Stack

Mappings

PLT

Heap

Code

0x0804800

0x?????????
Exploiting: ASLR – Memory Layout

- Stack
- glibc
- PLT
- Heap
- Code

system()
Exploiting: ASLR – ret2libc / ret2plt

• Defeats ASLR

• Also defeats DEP in one go 😊

• Just do:
  • EIP = &system@plt
  • arg = &meterpreter bash shellcode

• system(“/bin/bash nc –l –p 31337”)
Exploiting: ASLR

Other ASLR exploits:

- Partial RIP overwrite
- little endianness: \(0x11223344\)

```
<table>
<thead>
<tr>
<th>buf</th>
<th>44</th>
<th>33</th>
<th>22</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>52</td>
<td>33</td>
<td>22</td>
<td>11</td>
</tr>
</tbody>
</table>
```
Exploiting: ASLR

Other ASLR exploits:

- NOP sleds
  - As often used with JavaScript
  - Heap spray a few megabytes...

```
NOP NOP NOP NOP NOP NOP ... CODE
```

© Compass Security Schweiz AG
Recap: Anti ASLR

Anti-ASLR:

- Find static locations (like PLT)
- Mis-use existing pointers
- Spray & Pray
Recap! Exploit Mitigation Exploits

All three exploit mitigations can be defeated by black magic

Easily

Is there a solution?
The solution to all problems… PIE
Exploit Mitigation++

• **Fix:**
  • Compile as PIE
  • PIE: Position Independent Executable
  • Will randomize Code and PLT, too

• **Note:**
  • Shared libraries are PIC
    • (Position Independent Code)
  • Because they don’t know where they are being loaded
Exploiting: ASLR for code: PIE
Ok ok, everything is now ASLR’d and secure
Can I get my pizza now?
[the cake is a lie]
ASLR vs Information Leak

ASLR assumes attacker can’t get information

What if they can?

Meet: Memory Leak
ASLR vs Memory Leak

`char buf1[16] *ptr SFP EIP`

`send(socket, buf1, sizeof(int) * 16, NULL);`

Oups, attacker got 64 bytes back

- Pointer to stack, code, heap
- Can deduce base address
### ASLR vs Memory Leak

```c
send(socket, buf1, sizeof(int) * 16, NULL);
```

```plaintext
<table>
<thead>
<tr>
<th>char buf1[16]</th>
<th>*ptr</th>
<th>SFP</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```plaintext
<table>
<thead>
<tr>
<th>char buf1[16]</th>
<th>*ptr</th>
<th>SFP</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Exploiting: ASLR for code: PIE

Stack?

Real Stack

Real Heap

Heap?

Real Code

Code?

0xbfbfbfb

0xaabbccddd

0xdddeeffaa

0xbfbfbfbfbf
TL;DR

Enable ALL the mitigations (DEP, ASLR w/PIE, Stack Protector)

Defeat ALL the mitigations:

- ROP shellcode as stager to defeat DEP
- Information leak to defeat ASLR
- Non sIP-based stack-overflow vulnerability
## Comparison

<table>
<thead>
<tr>
<th>Stack Canary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Overflow</td>
</tr>
<tr>
<td>Inter-Chunk Heap Overflow</td>
</tr>
<tr>
<td>Arbitrary Write</td>
</tr>
<tr>
<td>Use After Free</td>
</tr>
<tr>
<td>Type confusion</td>
</tr>
</tbody>
</table>
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>DEP</th>
<th>ASLR</th>
<th>DEP + ASLR</th>
<th>PIE + DEP + ASLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shellcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shellcode + Heap Spray</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shellcode + Info Leak</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ret2libc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ret2plt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ret2plt + Infoleak</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROP + Info Leak</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
There are multiple Exploit Mitigations

All fail with the right vulnerability

But: They make exploit development harder

Somewhat

As shown in CTF’s at hacker cons, vulns can be identified and a reliable exploit developed in a few hours

- While being drunk
- Int3pids, dragon sector, shellphish, etc.
- Tomorrow, 17:30, Insomnihack Geneve
Linux Hardening
Exploit Mitigations

Enable DEP:

- Default since like forever
- (for old cpus: kernel.exec-shield = 1)
- To disable for a binary: gcc -z noexecstack

Enable ASLR:

- Default since like forever
- /proc/sys/kernel/randomize_va_space = 2

Enable Stack protector:

- fstack-protector (Default)
- fstack-protector-all (ALL Functions)
- fstack-protector-strong (Better)
Anti-Exploitation - Hardening

More Compiler options:

- **-D_FORTIFY_SOURCE=2**
  - FORTIFY_SOURCE provides (lightweight) buffer overflow checks for the following functions:
    - memcpy, mempcpy, memmove, memset, strcpy, stpcpy, strncpy, strcat, strncat, sprintf, vsprintf, snprintf, vsnprintf, gets.
  - **Compile time warnings**
    - **Default** in Ubuntu

- **Formatstring**
  - **Default** in Ubuntu
  - -Wformat -Wformat-security

- **Full Static Relocation:**
  - **Default** in Ubuntu
  - -Wl,-z-,relro -Wl,-z,now

- **Position independent code**
  - **NOT Default** in Ubuntu (*performance penalty*)
  - -pie –fPIE
# Ubuntu Packages Compiled as PIE

<table>
<thead>
<tr>
<th>Source package</th>
<th>8.04 LTS</th>
<th>9.04</th>
<th>9.10</th>
<th>10.04 LTS</th>
<th>10.10</th>
<th>11.04</th>
<th>11.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>openssh (native)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>apache2</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>bind9</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>opendap</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>postfix</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cups</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>postgresql-8.3</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>samba (native)</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>dovecot</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>dhcp3</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ntp</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>amavisd-new</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>squid</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cyrus-sasl2</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>exim4</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>nagios3</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>nagios-plugins</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>xinetd</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ipsec-tools</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>mysql-dfsg-5.1</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>evince</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>firefox</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>gnome-control-center</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>tiff</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>totem</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>qemu-kvm</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>yes</td>
</tr>
<tr>
<td>pidgin</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>yes</td>
</tr>
</tbody>
</table>
### Check: Checksec

<table>
<thead>
<tr>
<th>Process</th>
<th>RELRO Mode</th>
<th>Canary Status</th>
<th>NX Enabled</th>
<th>PIE Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>Full RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>dbus-launch</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>dbus-daemon</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>dbus-daemon</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>upstart-event-b</td>
<td>Full RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>window-stack-br</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>upstart-dbus-br</td>
<td>Full RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>upstart-dbus-br</td>
<td>Full RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>upstart-file-br</td>
<td>Full RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>ibus-daemon</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>unity-settings-b</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>bamfdaemon</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>at-spi-bus-laun</td>
<td>Full RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>gnome-session</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>dbus-daemon</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>gvfsd</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>ibus-dconf</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>ibus-ui-gtk3</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>ibus-x11</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>gvfsd-fuse</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>at-spi2-registr</td>
<td>Full RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>pulseaudio</td>
<td>Full RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>ibus-engine-sim</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>metacity</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>dconf-service</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>gnome-panel</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>indicator-appli</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>unity-fallback-</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>indicator-bluet</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>vmtoolsd</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>polkit-gnome-au</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>nautilus</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>nm-applet</td>
<td>Partial RELRO</td>
<td>Canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>initctl</td>
<td>Full RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>PIE enabled</td>
</tr>
<tr>
<td>indicator-messa</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
<tr>
<td>indicator-powerr</td>
<td>Partial RELRO</td>
<td>No canary found</td>
<td>NX enabled</td>
<td>No PIE</td>
</tr>
</tbody>
</table>
Check: Paxtest

Executable anonymous mapping  : Killed
Executable bss          : Killed
Executable data         : Killed
Executable heap         : Killed
Executable stack        : Killed
Executable shared library bss : Killed
Executable shared library data : Killed
Executable anonymous mapping (mprotect) : Vulnerable
Executable bss (mprotect)  : Vulnerable
Executable data (mprotect) : Vulnerable
Executable heap (mprotect) : Vulnerable
Executable stack (mprotect) : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect) : Vulnerable
Writable text segments : Vulnerable
Check: Paxtest

Anonymous mapping randomization test : 28 quality bits (guessed)
Heap randomization test (ET_EXEC) : 13 quality bits (guessed)
Heap randomization test (PIE) : 28 quality bits (guessed)
Main executable randomization (ET_EXEC) : 28 quality bits (guessed)
Main executable randomization (PIE) : 28 quality bits (guessed)
Shared library randomization test : 28 quality bits (guessed)
VDSO randomization test : 11 quality bits (guessed)
Stack randomization test (SEGMEXEC) : 28 quality bits (guessed)
Stack randomization test (PAGEEXEC) : 28 quality bits (guessed)
Arg/env randomization test (SEGMEXEC) : 20 quality bits (guessed)
Arg/env randomization test (PAGEEXEC) : 20 quality bits (guessed)
Randomization under memory exhaustion @~0: 28 bits (guessed)
Randomization under memory exhaustion @0 : 28 bits (guessed)
Return to function (strcpy) : return addr has NULL byte
Return to function (memcpy) : Vulnerable
Return to function (strcpy, PIE) : return addr has NULL byte
Return to function (memcpy, PIE) : Vulnerable
Advanced Linux hardening

The non-standard stuff
Grsecurity

Uses PaX

- Kernel patch
- Improved DEP and ASLR
- For userspace
- And kernelspace protection (e.g. SMAP emulation)
- Better randomness, more randomness

Also provides:

- Chroot hardening
- Hide /proc stuff
- Ptrace restrictions
- Kernel module loading restrictions
- RBAC (Role Based Access Control)
Ok I now know everything, pizza?
Container

Linux Container
Linux Container

Relevant?

- TEH CLOUD

Container: All container share the same kernel

- LXC
- Docker
- FreeBSD Jails (since March 2000)
- Solaris Zones
- Obsolete: Vserver, openvz

Virtualization: Each guest has his very own kernel

- Vmware, virtualbox, kvm, …
- Not covered here

RBAC’s

- SELinux (redhat), Apparmour (Suse), …
- Not covered here
! Container

- Chroot is not a container
  - Path restriction only
  - But: Can access other processes, the kernel, IPC, etc.

```bash
compass@ubuntu:~$ sudo chroot /var/chroot
root@ubuntu:/# cd root/
root@ubuntu:/root# ./w00t -0 --dir /nonexisting.clsic
[+] creating /nonexisting directory
[+] chrooting to /nonexisting
[+] change working directory to real root
[+] chrooting to real root
root@ubuntu:/# ls /
bin  cdrom  etc  initrd.img  lib64  media
boot  dev  home  lib  lost+found  mnt
root@ubuntu:/# 
```
LXC - Namespaces

LXC/Docker: Use namespaces for containerization
- Restrict view/access of certain processes

Linux provides the following namespaces:

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Constant</th>
<th>Isolates</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC</td>
<td>CLONE_NEWIPC</td>
<td>System V IPC, POSIX message queues</td>
</tr>
<tr>
<td>Network</td>
<td>CLONE_NEWNET</td>
<td>Network devices, stacks, ports, etc.</td>
</tr>
<tr>
<td>Mount</td>
<td>CLONE_NEWNS</td>
<td>Mount points</td>
</tr>
<tr>
<td>PID</td>
<td>CLONE_NEWPID</td>
<td>Process IDs</td>
</tr>
<tr>
<td>User</td>
<td>CLONE_NEWUSER</td>
<td>User and group IDs</td>
</tr>
<tr>
<td>UTS</td>
<td>CLONE_NEWUTS</td>
<td>Hostname and NIS domain name</td>
</tr>
</tbody>
</table>
LXC

Lxc container cannot:
- Interact with host processes
- Access root file system
- Access special devices (block, network, …)
- Mount filesystems
- Execute special ioctl’s

Lxc container can access:
- /proc: certain files
- /sys: certain files
- Do a lot of other stuff
LXC container share their Kernel…
Wait – what about Kernel security?

Userspace vs. kernelspace
Linux Kernel

Protecting the kernelspace
Contrary to popular belief, most vulns are not in device drivers.
Linux Kernel Syscalls

```plaintext
28 sys_rename
29 sys_mkdir
30 sys_rmdir
31 sys_dup
32 sys_pipe
33 sys_times
34 sys_prof [sys_nr_syscall]
35 sys_brk
36 sys_setgid
37 sys_getgid
38 sys_signal
39 sys_geteuid
40 sys_getegid
41 sys_acct
42 sys_umount2 [sys_umount] (2.2+)
43 sys_lock [sys_nr_syscall]
44 sys_ioctl
45 sys_fcntl
46 sys_mpx [sys_nr_syscall]
47 sys_setpgid
48 sys_ulimit [sys_nr_syscall]
49 sys_oldoldname
50 sys_umask
51 sys_chroot
52 sys_stat
53 sys_dup2
54 sys_getppid
55 sys_getpgid
56 sys_getgroups
57 sys_setsid
58 sys_sigtimedwait
59 sys_pivot_root
60 sys_getsid
61 sys_getresuid
62 sys_getresgid
63 sys_setresuid
64 sys_setresgid
65 sys_fchdir
66 sys_fchown
67 sys_fchmod
68 sys_fchmodat
69 sys_getsid
70 sys_setsid
71 sys_getsid
72 sys_time
73 sys_getitimer
74 sys_link
75 sys_unlink
76 sys_jointimes
77 sys_futimes
78 sys_mknod
79 sys_rmdir
80 sys_stat
81 sys_lstat
82 sys_fstat
83 sys_lchown
84 sys_lchown
85 sys_fchown
86 sys_lchown
87 sys_fchown
88 sys_dup_r
89 sys_pipe
90 sys捍
91 sys_creat
92 sys_open
93 sys_close
94 sys_read
95 sys_write
96 sys_write
97 sys_write
98 sys_write
99 sys_write
100 sys_write
101 sys_write
102 sys_write
103 sys_write
104 sys_write
105 sys_write
106 sys_write
107 sys_write
108 sys_write
109 sys_write
110 sys_write
111 sys_write
112 sys_write
113 sys_write
114 sys_write
115 sys_write
116 sys_write
117 sys_write
118 sys_write
119 sys_write
120 sys_write
121 sys_write
122 sys_write
123 sys_write
124 sys_write
125 sys_write
126 sys_write
127 sys_write
128 sys_write
129 sys_write
130 sys_write
131 sys_write
132 sys_write
133 sys_write
134 sys_write
135 sys_write
136 sys_write
137 sys_write
138 sys_write
139 sys_write
140 sys_write
141 sys_write
142 sys_write
143 sys_write
144 sys_write
145 sys_write
146 sys_write
147 sys_write
148 sys_write
149 sys_write
150 sys_write
151 sys_write
152 sys_write
153 sys_write
154 sys_write
155 sys_write
156 sys_write
157 sys_write
158 sys_write
159 sys_write
160 sys_write
161 sys_write
162 sys_write
163 sys_write
164 sys_write
165 sys_write
166 sys_write
167 sys_write
168 sys_write
169 sys_write
170 sys_write
171 sys_write
172 sys_write
173 sys_write
174 sys_write
175 sys_write
176 sys_write
177 sys_write
178 sys_write
179 sys_write
180 sys_write
181 sys_write
182 sys_write
183 sys_write
184 sys_write
185 sys_write
186 sys_write
187 sys_write
188 sys_write
189 sys_write
190 sys_write
```

Linux Kernel Protection Mechanisms

So, what about the Linux Kernel?

ASLR: **No**
- kASLR
- Since Kernel 3.14
- Disabled by default in most distributions
- Weaker than userspace (less entropy)
  - But: crash in kernel is very noticeable

DEP: **Yes**
- But some pages are W & X…
  - Because of X86 (BIOS etc.)
  - Therefore, not so useful

Stack Protector: **YES**

**FORTIFY_SOURCE**: **YES**
### Tools and Workarounds for Kernel Security

<table>
<thead>
<tr>
<th>Year</th>
<th>CPU Model</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>Ivy Bridge</td>
<td>i7 48xx, 49xx</td>
</tr>
<tr>
<td>2013</td>
<td>Haswell</td>
<td>i7 47xx</td>
</tr>
<tr>
<td>2014</td>
<td>Broadwell</td>
<td>i7 56xx, 55xx, 58xx, 59xx</td>
</tr>
<tr>
<td>2015</td>
<td>Skylake</td>
<td>i7 65xx, 66xx, 67xx, 68xx, 69xx</td>
</tr>
</tbody>
</table>

#### SMEP: Supervisor Mode Execution Protection
- Deny Kernel execution from userspace memory (ret2usr)
- Since Kernel 3.0
- Needs CPU support: Ivy Bridge ++
- Enabled by default in modern distributions
- Workaround: In-kernel ROP
- `cat /proc/cpuinfo | grep smep`

#### SMAP: Supervisor Mode Access Prevention
- Deny Kernel direct access to userspace memory
- Since Kernel 3.7
- Needs CPU support: Broadwell ++
- Enabled by default in modern distributions
From: Linus Torvalds <torvalds@linux-foundation.org>
Newsgroups: fa.linux.kernel
Subject: Re: [stable] Linux 2.6.25.10
Date: Tue, 15 Jul 2008 02:28:23 UTC
Message-ID: <fa.FocnvcnLqG7kPaYdYdPf5PbhjC@ifi.uio.no>

On Tue, 15 Jul 2008, pageexec@freemail.hu wrote:
>
> so guys (meaning not only Greg but Andrew, Linus, et al.), when will you
> publicly explain why you’re covering up security impact of bugs? and even
> more importantly, when will you change your policy or bring your process
> in line with what you declared?

We went through this discussion a couple of weeks ago, and I had
absolutely zero interest in explaining it again.

I personally don’t like embargoes. I don’t think they work. That means
that I want to fix things asap. But that also means that there is never a
time when you can "let people know", except when it’s not an issue any
more, at which point there is no _point_ in letting people know any more.

So I personally consider security bugs to be just "normal bugs". I don’t
cover them up, but I also don’t have any reason what-so-ever to think it’s
a good idea to track them and announce them as something special.

So there is no "policy". Nor is it likely to change.

Linus
On Tue, 15 Jul 2008, Linus Torvalds wrote:

> So as far as I'm concerned, "disclosing" is the fixing of the bug. It's
> the "look at the source" approach.

Btw, and you may not like this, since you are so focused on security, one reason I refuse to bother with the whole security circus is that I think it glorifies - and thus encourages - the wrong behavior.

It makes "heroes" out of security people, as if the people who don't just fix normal bugs aren't as important.

In fact, all the boring normal bugs are _way_ more important, just because there's a lot more of them. I don't think some spectacular security hole should be glorified or cared about as being any more "special" than a random spectacular crash due to bad locking.

Security people are often the black-and-white kind of people that I can't stand. I think the OpenBSD crowd is a bunch of masturbating monkeys, in that they make such a big deal about concentrating on security to the point where they pretty much admit that nothing else matters to them.

To me, security is important. But it's no less important than everything *else* that is also important!

Linus

http://yarchive.net/comp/linux/security_bugs.html

Linux Kernel Politics

Infrastructure people
   ✦ Don’t know they are there, except when something breaks

8 stable kernel trees!
   ✦ And Distros have their own stable kernels…

Actively hide security fixes in commit messages
   ✦ And they are honest about this

Distros in charge of security
   ✦ Is this good or not?

Conclusion:
   ✦ Important security fixes are maybe not backported to stable kernels
Kernel Hardening

Reduce features!

- Make menuconfig
- Remove:
  - Drivers
    - CVE-2016-2384: arbitrary code execution due to a double-free in the usb-midi linux kernel driver
  - Features
  - Protocols

Use grsecurity / PaX

Use current CPU

Enable kASLR
Kernel Hardening

Seccomp-bpf

- Seccomp: Since Kernel 2.6.12 (2005)
- Seccomp-bpf: Since Kernel 3.5 (2012)
- Whitelist (blacklist) system calls
  - E.g. exit(), read(), write(), …
- Who cares?
  - Chrome-Flash, Chrome-Renderer, vsftpd, OpenSSH, Firefox, Tor, …
Kernel Hardening

FS hardening

- /proc
- /sys
- /dev/[zero, null, urandom]
- Nothing else

AppArmour

- “additional restrictions on mounts, socket, ptrace and file access. Specifically restricting cross-container communication.”
Contrary to popular belief, most vulns are not in device drivers.
Recap! Linux Kernel

Container share same Kernel

Kernel is not very secure
Conclusion
Recap! Recap!

Want more security?
- Compile everything as PIE

Even more?
- Grsecurity Kernel patch

More Kernel security?
- Strip kernel of features
- Seccomp-bpf

-> Or better: Ask your distribution to do it! <-
Recap! Recap!

The good news:

Most companies get owned by web vuln’s anyway
  ✦ SQL injection
  ✦ Shell upload

Or social engineering...
Questions?
When I’m quick:
More slides (backup slides)
Advanced Hardening - CLANG CFI

Clang is a C/C++ frontend for LLVM

Has Control Flow Integrity!

- `-fsanitize=cfi`
- Mostly helps against type confusion attacks

- `-fsanitize=cfi-cast-strict`: Enables strict cast checks
- `-fsanitize=cfi-derived-cast`: Base-to-derived cast to the wrong dynamic type.
- `-fsanitize=cfi-unrelated-cast`: Cast from void* or another unrelated type to the wrong dynamic type.
- `-fsanitize=cfi-nvcall`: Non-virtual call via an object whose vptr is of the wrong dynamic type.
- `-fsanitize=cfi-vcall`: Virtual call via an object whose vptr is of the wrong dynamic type.
- `-fsanitize=cfi-icall`: Indirect call of a function with wrong dynamic type.
Hipster exploiting

Blind ROP
- Brute force all ROP gadgets
- Based on replies (Crash, Freeze, No Crash)
- No need to know anything about the process (don’t even need binary!)

Sigreturn oriented programming
- Use signal handler to invoke code
- Does not need as many gadgets as normal ROP (just “syscall; ret”)

Clang Compiler

Frontend for LLVM (compiles C to LLVM IR)

SafeStack
- `-fsanitize=safe-stack`
- Split stack into safe- (SIP etc.) and unsafe stack
Clang Compiler: Address Sanitizer

- Detects memory corruption bugs
- Heavy performance penalty
- Do not use in production!

Date: Wed, 17 Feb 2016 23:19:21 +0100
From: Szabolcs Nagy <nsz@...t70.net>
To: oss-security@...ts.openwall.com
Subject: Address Sanitizer local root

There is an alarming trend that Address Sanitizer and related compiler instrumentations from compiler-rt are used as a hardening solution and run in production.