ROP is Still Dangerous: Breaking Modern Defenses

Nicholas Carlini, David Wagner
University of California, Berkeley
History: Memory Corruption

- Cat and mouse game! (Attackers VS Defenders)

- StackGuard
- StackShield
- ProPolice
- StackGhost

Stack Overflow, Format String

Canary/Shadow stack

Heap/Integer Overflow

W⊕X

Return to libc, ROP

ASLR

ROP+

Getting more sophisticated to evade existing defenses

Code injection

No need of code injection

Bypass ASLR
Gadgets

- Variable-Length nature of x86 instruction set

Unintended / Unaligned gadget (overlapped) Intended / Aligned gadget

Text(code) section

<table>
<thead>
<tr>
<th>Program code</th>
<th>4A8223C7</th>
<th>mov al,0x1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4A8223C9</td>
<td>cmp al,bl</td>
</tr>
<tr>
<td></td>
<td>4A8223CB</td>
<td>lea eax,[ebp-0x80]</td>
</tr>
</tbody>
</table>

| Source: “Smashing the gadgets” by Vasilis and Michalis [SP’12] |
Return-Oriented Programming (1/2)

Source: kBouncer presented by Vasilis (USENIX ’13)
Return-Oriented Programming (2/2)

- **Goal:** control the flow to run arbitrary code!
  - Grab arbitrary codes (gadgets) of an attacker’s choice
  - Build multiple gadgets chained together (stack pivot)
  - Execute them w/ esp control (instead of eip)

- **Characteristics**
  - Reuse the existing code w/ executable permission
  - Use chained gadgets and write them on the stack beforehand

- **ROP**
  - Turing-complete computation
  - Generalization of return-into-libc
Common Defense

- ASLR (Address Space Layout Randomization)
  - At Runtime
  - Randomly moving the segments(sections)

- But still common in the wild
  - ASLR-disabled modules
  - Information disclosure vulnerability
    (JIT code reuse: On the effectiveness of fine-grained ASLR, SP’13)
  - Low entropy of randomness in 32 bits
Fine-Grained ROP Defenses in Research

- Compiler-based
- Binary Instrumentation
- CFI (Control Flow Integrity)
- Others
Various Approaches in Research

ROP is Still Dangerous

Performance Overhead

High

ROPdefender [AsiaCCS’11]

HyperCrop [ICISS’11]

DROP [ICISS ’09]

Low

kBouncer [USENIX’13]

ROPPecker [NDSS’14]

IPR (Orp) [S&P’12]

Binary Stirring [CCS’12]

CCFIR [S&P’13]

G-Free [ACSAC’10]

Return-less [EuroSys’10]

CFLocking [ACSAC’11]

Requirements

Program binary

Binary rewriting

Source code

Source: ROPPecker presentation by Yueqiang Cheng (NDSS ’14)
Attack targets: state-of-the-art ROP defenses

- \textit{kBouncer} by Vasilis et al.
- \textit{ROPecker} by Yueqiang et al.
LBR (Last Branch Record)

- Dedicated CPU registers
  - Hardware feature (by Intel)
  - Filters certain types of branches (relative/indirect calls/jmps/rets)
  - Allows for inspecting the last 16 indirect branches
  - Stores the last 16 executed branches in MSR

- Both kBouncer and ROPecker leverage LBR
- Illegal returns
  - ROP issues returns to non-call-preceded addresses
- Long “short gadgets”
  - Payloads built with long sequences of short gadgets

Source: kBouncer presentation by Vasilis (USENIX ’13)
**ROPecker: Idea / Defense Key**

- Builds on ideas found in *kBouncer*
  - ROP issue returns to non-call-preceded addresses
  - Payloads built with long sequences of short gadgets

- Keep track of executable set (pages marked executable)

- Check if there is a long chain of gadget-like sequences (threshold: normal max=10, ROP min=17)

\[
\text{Max}_{\text{normal}} < \text{Detection} \leq \text{Min}_{\text{ROP}}
\]

- Invoke when critical system calls (i.e. mmap, mprotect, execve, ...)

ROP is Still Dangerous
Evaluation: *kBouncer & ROPecker*

- **Generic**
  - All types of ROP gadget: ret-based, jmp-based, ...

- **Efficient**
  - Runtime overhead (1-4% at most)

- **Transparent**
  - No source code or compiler, No binary instrumentation

- **Limitation**
  - Lacks flexibility: monitoring only 16 branches by design
Key Attack Primitives using weaknesses (1/4)

- **Call-Preceded ROP**
  - Both defenses check if gadget is non-call-preceded!

- **Evasion Attack**
  - Both defenses use a length-based classifier!

- **History Flushing**
  - Both defenses keep only a limited amount of history inspection!
  - Moreover, not constantly but periodically! (e.g. system call)
Use call-preceded gadgets (6%)
(All gadgets start at a call-preceded address)

```
and  [rax], 0xf
mov  edx, 0x768
mov  esi, 0x4ab632
mov  rdi, rbx
    push rbx
mov  rdi, rbx
    mov ebx, eax
call 0x2b2130
    add ebx, ebx
test rbp, rbp
    add ebx, eax
cmov [rbp], 0x0
    pop rbx
add  rsp, 0x8
pop  rbx
pop  rbp
ret
```

Source: SOP is still Dangerous presented by Nicholas (USENIX ’14)

Intuitively, this might be hard to perform ROP, but...
- 70KB of binary code was sufficient to mount full ROP attacks!
Recall *kBouncer* & *ROPecker* defenses...
- A length-based classifier
- Heuristics: Long sequences of short gadgets

How to defeat monitoring runtime behavior?
- Find gadgets that look like benign execution!
- Use long gadgets!!
- Use a mixture of both short and long gadgets for evasion!!!
Recall *kBouncer* & *ROPecker* defenses...
- Limited amount of history inspection (e.g. LBR size)
- Periodical checks for performance consideration

How to fool monitoring history inspection?
- Use gadgets to hide history!
- Perform ROP attack when not being watched!!
- Wipe the history clean of any evidence of past payloads, before defender’s inspection process is invoked!!!
- During inspection time, insert no-op instruction!!!!
Goal: Issue a single syscall

i.e. `mprotect()` on Linux, `VirtualProtect()` on Windows

Threat model assumptions

- An attacker has a known exploit to allow control IP
- `WX/ASLR` enabled
- Program contains at least one ASLR-disabled library
- An attacker knows the defense is present and how it works
**Attack against kBouncer (1/4)**

- kBouncer has two checkpoints (system call is invoked)
  - Call-preceded?
  - Are the latest branches gadget-like?

Source: *SOP is still Dangerous* presented by Nicholas (USENIX ’14)
History Hiding Attack
Uses history flushing to clear evidence of ROP

Attack steps
1. Initial exploitation (using any gadgets):
   - Normal ROP attack until a syscall is about to be invoked
2. Hide the history by flushing:
   - Short flushing gadget (call-preceded) until LBR contains them only
   - Long termination gadget with at least 20 instructions
3. Restore registers:
   - Pop values off from the stack (call-preceded)
4. Issue the system call
Attack against *kBouncer* (3/4)

- Check the LBR to see if
  - Call-preceded? Yes → Good!
  - Long chain? No → Good!

Source: *SOP is still Dangerous* presented by Nicholas (USENIX ’14)
Would it be better or effective if
- *kBouncer* has complete view of history?
- LBR with infinitive entries?

Yes, it would break history flushing attack, but...

Evasion attack steps

Use call-preceded gadgets only in initialization
(1) (call-preceded) setup
(2) (call-preceded) termination gadget
(3) (call-preceded) register restoration
(4) Invoke syscall
Phases: Repeated Hiding Attack

(1) Initialization: Insert a termination gadget
- This will stop ROPecker from looking further back in LBR

(2) Loading: Load call-preceded gadgets + termination gadget
- ROPecker will run before each page load gadget is invoked

Source: SOP is still Dangerous presented by Nicholas (USENIX ’14)
Attack against *ROPecker* (2/3)

- **Phases: Repeated Hiding Attack**
  1. **(3) Attack:** Use any gadgets on loaded pages to mount an attack
     - Attack can be distributed among multiple phases
  2. **(4) History Hiding:** Invoke a shot flushing gadget to fill LBR
     - When *ROPecker* next runs, it will see no attack prior to this point

---

Source: *SOP is still Dangerous* presented by Nicholas (USENIX ’14)
Evasion attack

- Letting ROPecker inspect at any time

- Insert a termination gadget in between every ten useful gadgets

- Threshold = 11
Conclusion

- Implication for defenses
  - Do not rely on limited amount of history
  - Choosing call-preceded ROP (6%) is feasible (in >70K text)
  - Classifying code as “gadget” vs “non-gadget” is challenging
  - CFI needs to return to its root

- What is fundamental properties to determine ROP attacks?