Hacking Blind

Andreas Amler

Control Flow Integrity Based Security
21.11.2016
Overview

ROP Introduction

Blind ROP
  Attack Outline
  Finding gadgets
  Finding the PLT

Evaluation and Defenses
  Number of requests
  Protecting against BROP
The ROP Attack

Scenario:
- We want to attack a server application
- We have a copy of the binary
- We found a buffer overflow
Buffer overflows

Function’s stack layout:

<table>
<thead>
<tr>
<th>buffer</th>
<th>local variables</th>
<th>saved frame pointer</th>
<th>saved return address</th>
<th>...</th>
</tr>
</thead>
</table>

Buffer overflow:

<user input>
Buffer overflows

Function’s stack layout:

<table>
<thead>
<tr>
<th>buffer</th>
<th>local variables</th>
<th>saved frame pointer</th>
<th>saved return address</th>
<th>...</th>
</tr>
</thead>
</table>

Buffer overflow:

<table>
<thead>
<tr>
<th>buffer</th>
<th>&lt;user input&gt;</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>saved return address</td>
</tr>
</tbody>
</table>
How to use the overflow?

Ideas:
- We can jump to any memory address
- We want to spawn a shell
- Without NX: Just inject shell code
- With NX: Need to reuse existing code
How to use the overflow?

Ideas:

- We can jump to any memory address
- We want to spawn a shell
- Without NX: Just inject shell code
- With NX: Need to reuse existing code

Reusing code:

- Goal: execute system("/bin/sh");
- system() and the string are part of libc
- Use gadgets to supply arguments
The use of gadgets

What is a gadget?

- A code snippet that exists in the binary
- Ends with a ret-instruction
The use of gadgets

What is a gadget?
- A code snipped that exists in the binary
- Ends with a ret-instruction

Example:
```
pop rsi;
xor rax rax;
ret;
```
x86_64 Linux calling conventions

Put arguments into:
1. rdi
2. rsi
3. rdx
4. ...
x86_64 Linux calling conventions

Put arguments into:

1. rdi
2. rsi
3. rdx
4. ...

What does this mean for ROP?

- Gadgets: Get arguments from stack into registers
- Example for first argument:

```
pop rdi;
ret;
```
Example ROP chain

Stack
- saved return address
- address 1
- address 2
- address 3

Memory
- pop rdi;
- ret;
- "/bin/sh"
- system()
Overview

ROP Introduction

Blind ROP
  Attack Outline
  Finding gadgets
  Finding the PLT

Evaluation and Defenses
  Number of requests
  Protecting against BROP
BROP: The idea

As presented in *Hacking Blind* by A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières and D. Boneh [1]

Situation:

- We found a buffer overflow
- Vulnerable application restarts after a crash
- We do not have the binary
BROP: The idea

As presented in *Hacking Blind* by A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières and D. Boneh [1]

Situation:
- We found a buffer overflow
- Vulnerable application restarts after a crash
- We do not have the binary

Approach:
- Find a way to use the write syscall
- Dump the binary
- Proceed with regular ROP
The write syscall

Naive attempt:

```
pop rdi; ret; (arg 1: socket fd)
pop rsi; ret; (arg 2: address)
pop rdx; ret; (arg 3: length)
pop rax; ret; (syscall number)
syscall
```
Optimizations

Controlling rdx:

- `pop rdx` gadgets are rare
- Use `strcmp` instead
- `strcmp` sets rdx to string length
- `strcmp` can be found in the PLT
Optimizations

Controlling rdx:
- *pop rdx* gadgets are rare
- Use *strcmp* instead
- *strcmp* sets rdx to string length
- *strcmp* can be found in the PLT

Replacing the syscall:
- Find *write* in the PLT
- Replace *pop rax; syscall* with function call
Optimized procedure: Summary

1. Find gadgets
   - `pop rdi; ret`
   - `pop rsi; ret`

2. Find the PLT
   - Find entry for `write`
   - Find entry for `strcmp`
Finding gadgets

Basic idea:

- Try every address in .text segment
- Starting address without PIE: 0x400000
- Observe program behavior
Finding gadgets

Basic idea:

- Try every address in .text segment
- Starting address without PIE: 0x400000
- Observe program behavior

Most likely behaviors:

- Program crashes (trap)
- Program hangs (stop)
Finding gadgets

Basic idea:
- Try every address in .text segment
- Starting address without PIE: 0x400000
- Observe program behavior

Most likely behaviors:
- Program crashes (trap)
- Program hangs (stop)

Goal:
- Make the program not crash on gadget
Gadget finding example

- Scenario 1: Only replace return address

```
pop rdi;
ret;
crash
```

```
return address 0xdead
arbitrary value
arbitrary value
0xdead
```

Diagram:
- Stack: return address 0xdead 0xdead
- Memory: pop rdi; ret; crash
Gadget finding example

- Scenario 2: Using stop gadget
Finding different gadgets

Problem:
- Some gadgets pop more than 1 value
- How to identify them?
Finding different gadgets

Problem:
- Some gadgets pop more than 1 value
- How to identify them?

Solution:
- Use different patterns of *trap* and *stop*
Example: 3-pop gadget

```
Stack
return address 0xdead 0xdead 0xdead (stop)

Memory
pop rdi; pop rsi; pop rax; ret;

sleep
return address 0xdead 0xdead (stop)
```
Identifying gadgets

Problem:

- We cannot tell which registers were popped
Identifying gadgets

Problem:
  ▶ We cannot tell which registers were popped

Possible solution:
  ▶ Test different system calls
  ▶ Observe behavior
Optimization: The BROP gadget

Special properties:
- Pops 6 registers
- Present in every binary
- Parsed at offset 0x7: Different gadget
- Parsed at offset 0x9: Yet another gadget
Hacking Blind

Andreas Amler

ROP Introduction

Blind ROP

Attack Outline

Finding gadgets

Finding the PLT

Evaluation and Defenses

Number of requests

Protecting against BROP

BROP gadget illustration

```
pop rbx;
pop rbp;
pop r12;
pop r13;
pop r14;
pop r15;
ret;
pop rsi;
pop r15;
ret;
pop rdi;
ret;
```

```
0x0
0x7
0x9
```

```
pop rbx;
pop rbp;
pop r12;
pop r13;
pop r14;
pop r15;
ret;
```
Finding the BROP gadget

Is the gadget at address A?

1. Check for 6-pop gadget at A
2. Check for 2-pop gadget at A + 0x7
3. Check for 1-pop gadget at A + 0x9
The Procedure Linking Table (PLT)

- Stores code snippets for function calls
- Handles every external call (e.g. to libc)
The Procedure Linking Table (PLT)

- Stores code snippets for function calls
- Handles every external call (e.g. to libc)

\[ \text{call write} \]
\[ \ldots \]
\[ \text{jmp [write]} \]
\[ \text{push 0} \]
\[ \text{jmp dlresolve} \]
\[ \text{6 bytes} \]
\[ 16 \text{ bytes} \]
\[ \text{jmp [strcmp]} \]
\[ \text{push 2} \]
\[ \text{jmp dlresolve} \]
The PLT: Properties

Observations:

- Entries are 16 byte aligned
- Most PLT entries: EFAULT instead of crash
- Any entry offset by 0x6: no crash
The PLT: Properties

Observations:
- Entries are 16 byte aligned
- Most PLT entries: EFAULT instead of crash
- Any entry offset by 0x6: no crash

Searching for the PLT:
- Try addresses that are 16 bytes apart
- Check for crash
- Try addresses offset by 0x6
- Check for crash again
Finding write in the PLT

General approach:

- Test if a given PLT entry writes something
- Problem: How to find the correct file descriptor?
Finding write in the PLT

General approach:
- Test if a given PLT entry writes something
- Problem: How to find the correct file descriptor?

Possible solutions:
1. Try multiple writes in a ROP chain
2. Open multiple connections and guess
Finding strcmp in the PLT

Test behavior:

- `strcmp(bad,bad): crash`
- `strcmp(bad,readable): crash`
- `strcmp(readable,bad): crash`
- `strcmp(readable,readable): no crash`
**BROP attack conclusion**

We can now

- Use gadgets to set \textit{rdi} and \textit{rsi}
- Use \textit{strcmp} to set \textit{rdx}
- Use \textit{write} to dump the binary
- Proceed with regular ROP
Overview

ROP Introduction

Blind ROP
  Attack Outline
  Finding gadgets
  Finding the PLT

Evaluation and Defenses
  Number of requests
  Protecting against BROP
How long does BROP take?

Example application:

- Open-source SSL library
- Known stack vulnerability (yaSSL)
- Target: Older version of MySQL using yaSSL
- Mimics application with open-source component
How long does BROP take?

<table>
<thead>
<tr>
<th>Attack phase</th>
<th>Number of requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack reading</td>
<td>406</td>
</tr>
<tr>
<td>find PLT</td>
<td>1454</td>
</tr>
<tr>
<td>find BROP gadget</td>
<td>1481</td>
</tr>
<tr>
<td>find strcmp</td>
<td>1545</td>
</tr>
<tr>
<td>find write</td>
<td>1602</td>
</tr>
<tr>
<td>dump bin and exploit</td>
<td>3851</td>
</tr>
</tbody>
</table>

- Total number of requests: 10339
- Total time: 20 min
Protection: Sleep on crash

- Delay fork after a segfault
- Advantage: Slows down BROP a lot
- Disadvantage: Bugs can be used as DoS
Protection: Compiler methods

- Some compilers can insert bound checks
- Advantage: Prevents buffer overflows (not just ROP)
- Disadvantage: Bad performance (up to 2x slowdown)
Protection: Rerandomization

Weakness of BROP:

- Has to crash a process and re-connect many times
- Relies on gathered information staying the same
Protection: Rerandomization

Weakness of BROP:
- Has to crash a process and re-connect many times
- Relies on gathered information staying the same

How to use this?
- Call `exec` after `fork` on crash or spawn
- Causes ASLR to rerandomize address layout
Protection: Rerandomization

Weakness of BROP:
- Has to crash a process and re-connect many times
- Relies on gathered information staying the same

How to use this?
- Call `exec` after `fork` on crash or spawn
- Causes ASLR to rerandomize address layout

Assumed disadvantages:
- Requires code restructuring
- Runtime overhead
Evaluation of BROP

Advantages:

- Does not require a copy of the binary
- Defeats NX and ASLR (without rerandomization)
- Defeats 'security through obscurity'
Evaluation of BROP

Advantages:
- Does not require a copy of the binary
- Defeats NX and ASLR (without rerandomization)
- Defeats 'security through obscurity'

Disadvantages:
- Relatively complex
- Potentially high runtime
- Weak to rerandomization
Thanks for your attention!

Any questions?
A. Belay, A. Bittau, A. Mashtizadeh, D. Mazières, and D. Boneh.

Hacking blind.