Undermining Information Hiding (And What to do About it)

Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, Herbert Bos
Overview

• Mitigating code-reuse attacks
• Hiding code pointers in sensitive regions (information hiding)

• Sensitive region per thread
• Thread Spraying
• Reveal SafeStack (LLVM) in seconds

• Authenticating Page Mapper: **harden** information hiding
ROP attack

HEAP = dynamic data
STACK = program code execution context (contains code pointers)
ROP attack

HEAP = dynamic data
STACK = program code execution context  
(contains code pointers)

Inject FAKE STACK through normal data input
ROP attack

HEAP = dynamic data
STACK = program code execution context
(contains code pointers)

Inject FAKE STACK through normal data input

Hijack control flow through mem. vuln.
ROP attack

HEAP = dynamic data
STACK = program code execution context
(contains code pointers)

Inject FAKE STACK through normal data input

Hijack control flow through mem. vuln.

How does the attacker get to know the code pointers?
ROP attack

HEAP = dynamic data
STACK = program code execution context (contains code pointers)

Inject FAKE STACK through normal data input

Hijack control flow through mem. vuln.

How does the attacker get to know the code pointers?

- Fingerprint app version
Protect code pointers

- Address Space Layout Randomization

Leak pointer to CODE or DATA
Protect code pointers

- Address Space Layout Randomization
  - Leak pointer to CODE or DATA

- Function / Instruction Level Randomization
  - JIT-ROP
Protect code pointers

- Address Space Layout Randomization
  - Leak pointer to CODE or DATA

- Function / Instruction Level Randomization
  - JIT-ROP

- Isolate code pointers to a safe area
  - Hardware Segmentation
  - Software-Fault Isolation
  - Information Hiding
Isolating code pointers

• ASLR-Guard and Code Pointer Integrity

• Two types of safe areas
  • Safe Stack: code pointers located on the stack (like return addresses)
  • Safe Region: all other code pointers (like function addresses)

• Safe Stack is in production compiler LLVM
Information Hiding

• Preferred model because
  • Hardware Segmentation not available on 64 bit
  • Information Hiding lower perf. overhead than Software-Fault Isolation

• Information Hiding idea:
  • Separate Code Pointers to safe area
  • Assumes no pointers to safe area
  • Assumes high entropy of safe area

See paper
Our focus
Entropy

• Degree of randomness
• Given in bits

• Example:
  • 3 bit address space
  • 8 blocks of 1 byte

• Hide sensitive data

Sens. data: 2 bytes

$(2^1)$

000
001
010
011
100
101
110
111

Entropy: 2 bits

Hit chance: \( \frac{1}{2^2} = \frac{1}{4} \)

Worst case: #probes \( 2^2 = 4 \)
Entropy

• Degree of randomness
• Given in bits

• Example:
  • 3 bit address space
  • 8 blocks of 1 byte

• Hide sensitive data

Sens. data:

2 bytes
\(2^1\)

4 bytes
\(2^2\)

Entropy:

2 bits

1 bit

Hit chance:

\[\frac{1}{2^2} = \frac{1}{4}\]

\[\frac{1}{2^1} = \frac{1}{2}\]

Worst case:

#probes

\(2^2 = 4\)

\(2^1 = 2\)
64 bit address space

Entropy: 64 bits

Hide: 1 byte
64 bit address space

Linux user space only uses 47 bit

Entropy: 47 bits

Hide: 1 byte
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = \(2^{12}\) bytes

Hide: 4096 bytes

Entropy: 35 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

Hide: $2^{23}$ bytes

Entropy: 24 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**
Legitimately spawn as many threads as possible

Hide: $2^{23}$ bytes

Entropy: 24 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**

Legitimately spawn as many threads as possible

Spawn a new thread

Hide: $2^{24}$ bytes

Entropy: 23 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**
Legitimately spawn as many threads as possible

Spawn a new thread

Spawn 2 more threads

Hide: $2^{25}$ bytes

Entropy: 22 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**
Legitimately spawn as many threads as possible

Spawn a new thread

Spawn 2 more threads

Spawn 128k threads = $2^{17}$ stacks

Hide: $2^{40}$ bytes

Entropy: 7 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**
Legitimately spawn as many threads as possible

Spawn a new thread

Spawn 2 more threads

Spawn 128k threads = $2^{17}$ stacks

Drops worst case #probes to 128

Hide: $2^{40}$ bytes

Entropy: 7 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**
Legitimately spawn as many threads as possible

Spawn a new thread

Spawn 2 more threads

Spawn 128k threads = $2^{17}$ stacks

Mmap entropy is 40 bit => worst case #probes is 1 ($2^0$)

Entropy: 7 bits

Hide: $2^{40}$ bytes
Thread Spraying

• Firefox: unbounded JS web worker threads
• Chrome: about 250 JS web workers threads
• MySQL: about 1000 connection threads
Thread Spraying

• Firefox: unbounded JS web worker threads
• Chrome: about 250 JS web workers threads
• MySQL: about 1000 connection threads

FF is going to look at it and put a limit
Thread Spraying

• Firefox: unbounded JS web worker threads
• Chrome: about 250 JS web workers threads
• MySQL: about 1000 connection threads

• Proof-of-Concept in Firefox:
  • Spawn 2k threads
  • Crash-less memory probing
  • Find safe stack < 3 seconds

FF is going to look at it and put a limit
Authenticating Page Mapper

• Based on observations:
  • Active stack space is smaller than its actual size
  • Well defined access pattern

• Authenticates first access to registered pages

• Installs user-level page-fault handler
Authenticating Page Mapper

- Stack (8MB)
  - low addr
- Inactive
- Active
  - stack_base
  - high addr
- RSP
Authenticating Page Mapper

\[
\text{sub } 0x80, \%\text{rsp}
\]
sub 0x80, %rsp
mov 0x1000, (%rsp)
Authenticating Page Mapper

sub 0x80, %rsp
mov 0x1000, (%rsp)

page-fault handler checks:
RSP <= page-fault addr < stack_base
Authenticating Page Mapper

```
sub 0x80, %rsp
mov 0x1000, (%rsp)
```

diagram:
- Stack (8MB)
  - Active
  - Inactive

page-fault handler checks:
RSP <= page-fault addr < stack_base

OK
Authenticating Page Mapper

sub 0x80, %rsp
mov 0x1000, (%rsp)

page-fault handler checks:
RSP <= page-fault addr < stack_base
Authenticating Page Mapper

```
sub  0x80, %rsp
mov  0x1000, (%rsp)
```

page-fault handler checks:

\[ \text{RSP} \leq \text{page-fault addr} < \text{stack_base} \]

NOT OK
Authenticating Page Mapper

Attacker can follow pointers and get to the base, or get to active region by probing from the high addr.
Authenticating Page Mapper

Attacker can follow pointers and get to the base, or get to active region by probing from the high addr

=> Inflate stack and move stack into inflated area
Authenticating Page Mapper

Attacker can follow pointers and get to the base, or get to active region by probing from the high addr

=> Inflate stack and move stack into inflated area
Authenticating Page Mapper

Attacker can follow pointers and get to the base, or get to active region by probing from the high addr

=> Inflate stack and move stack into inflated area

=> Check remains the same
=> Trap has different handler and access is never allowed
Authenticating Page Mapper

Attacker can follow pointers and get to the base, or get to active region by probing from the high addr

=> Inflate stack and move stack into inflated area

=> Check remains the same

=> Trap has different handler and access is never allowed

=> Increases detection rate
Overhead APM

• Ran SPEC CPU2006 + Safe Stack **without and with** APM
  • 0.0% overhead*

• Ran browser benchmarks in Chrome and Firefox **without and with** APM
  • Chrome: 0.0% overhead*
  • Firefox: 0.5% overhead*

* = geometric mean
Worst case Detection Guarantees with APM

- Assumes attacker fills up Active Mem
- Active mem == Stack Size
- Detection Guarantees = \( (1 - \frac{Active\ Mem}{Stack\ Size \times inflation\ factor}) \times 100 \)

<table>
<thead>
<tr>
<th>Inflation factor</th>
<th>Detection Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>0%</td>
</tr>
<tr>
<td>2x</td>
<td>50%</td>
</tr>
<tr>
<td>4x</td>
<td>75%</td>
</tr>
<tr>
<td>8x</td>
<td>87.5%</td>
</tr>
<tr>
<td>10x</td>
<td>90%</td>
</tr>
</tbody>
</table>
APM limitations

• Application implementation issues
  • Pointer to active region of stack

• Determine active region through side channel attacks
Conclusion

• Demonstrated an efficient way to locate the Safe Stack through a new attack vector named Thread Spraying

• Stronger isolation techniques should be preferred over Information Hiding

• APM is a possible solution to harden Information Hiding until SFI is or can be widely deployed