Lec 11: Memory Disclosure

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Memory Disclosure

Memory Disclosure $\neq$ Memory Corruption

Memory disclosure can be caused by memory corruption, but memory disclosure does ***not*** necessarily involve memory corruption.

Buffer Over-Read

Buffer over-read is a bug that allows an attacker to read beyond the size of a buffer.

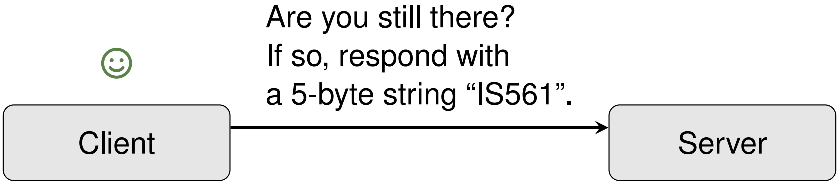
> Buffer over-read does ***not*** corrupt memory!

Example: Hearbleed Bug (in 2014)

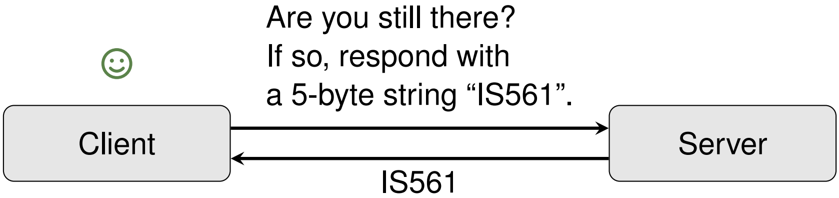
- Famous bug in OpenSSL (in TLS ***heartbeat***).
- An attacker could steal private keys.

Heartbleed Bug: High-level Idea

Heartbleed Bug: High-level Idea

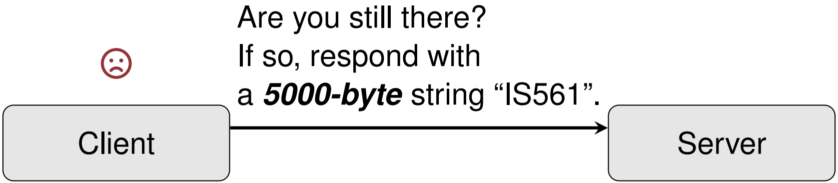


Heartbleed Bug: High-level Idea

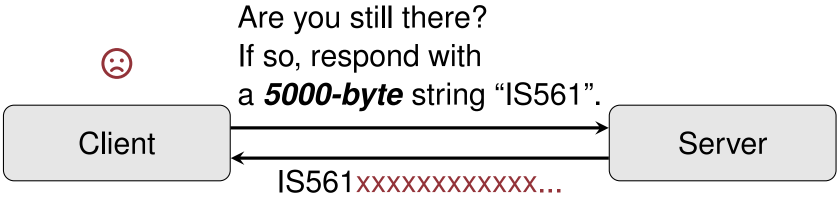


Heartbleed Bug: High-level Idea

Heartbleed Bug: High-level Idea



Heartbleed Bug: High-level Idea



The Bug

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length; // not necessarily the same as the SSL3_RECORD's length
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;

struct {
  unsigned int length; // length of the data
  unsigned char *data; // points to HeartbeatMessage
  ...
} SSL3_RECORD;

...

memcpy(bp, pl, length); // pl = HeartbeatMessage payload
                        // length = obtained from SSL3_RECORD
```

Other Memory Disclosure

- Format string vulnerability also leaks memory info.
  - `%08x.%08x.%08x...`
- Memory corruption bugs may allow memory leak.
  - E.g., Overwriting the length field of a string object.

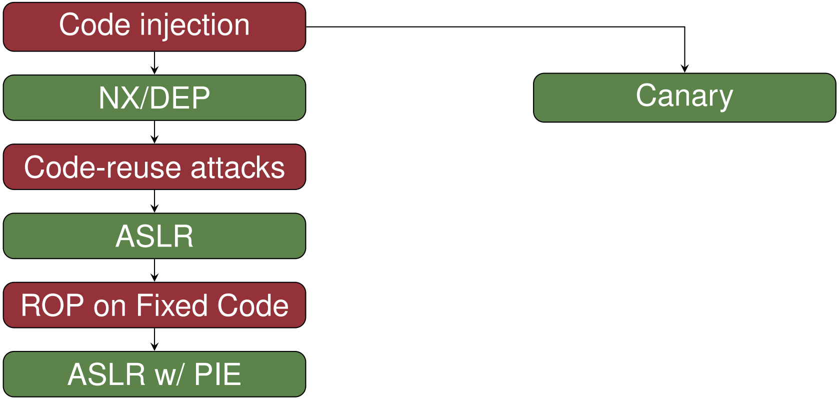
Memory Disclosure and Exploit

- It is possible that a program may have more than a single vulnerability.
  - For example, one memory corruption and one memory disclosure.
- In such a case, we can bypass existing defenses.
  - Canary bypass: canary value could be leaked.
  - ASLR bypass: code/stack pointers could be leaked.

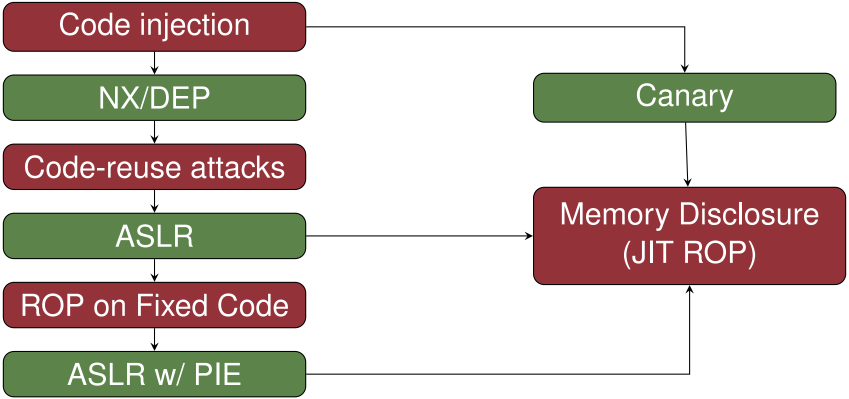
> Caveat: we should be able to leak memory contents and trigger the memory corruption within the same process.

JIT ROP

Attack/Defense So Far ...



Attack/Defense So Far ...



Just-In-Time ROP (JIT ROP)[1]

Generalization of combining memory disclosure and memory corruption exploits.

[1] Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization, ***Oakland 2013***.

JIT ROP Overview

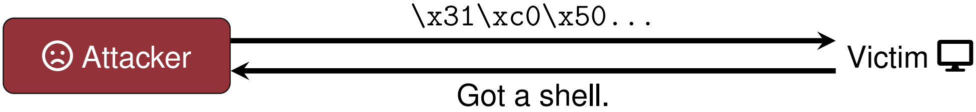
- Use a memory disclosure bug to get the code.
  - Assumption: there is a leaked function pointer that can be ***repeatedly*** used to read arbitrary memory addresses.
- Find ROP gadgets.
- (JIT) Compile ROP program for exploitation.

Harvesting Code Pages w/o a Crash?

- Leak a code page from a function pointer.
- Disassemble the page and ***recursively*** follow jump targets to discover more pages.

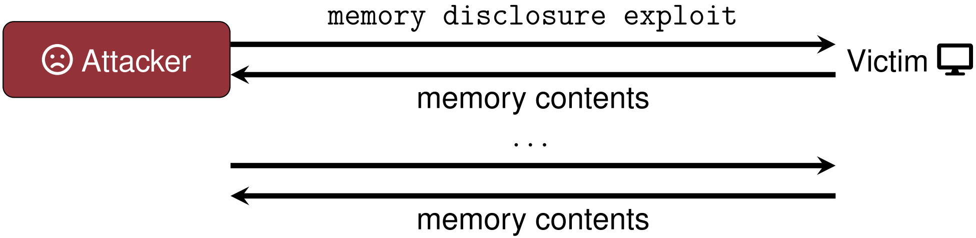
Traditional Exploit Development

1. Analyze the target binary offline.
2. Develop a control-hijack exploit.
3. Launch the exploit.



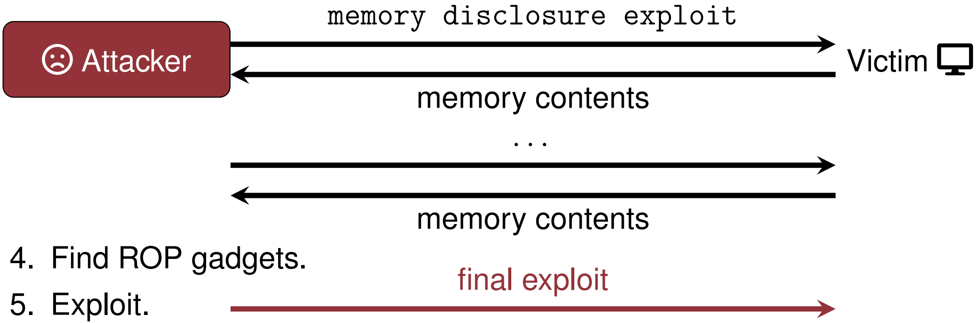
JIT ROP Exploitation

1. Analyze the target binary offline.
2. Develop a memory disclosure exploit.
3. Harvest code pages with the exploit.



JIT ROP Exploitation

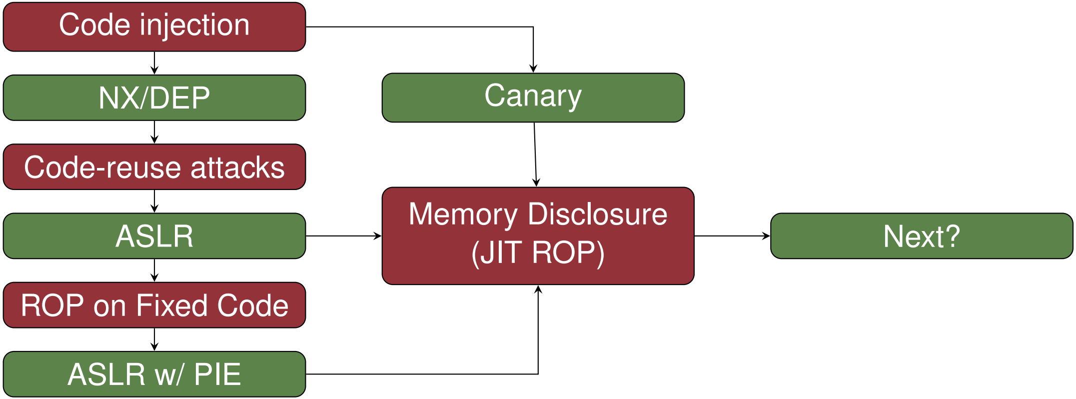
1. Analyze the target binary offline.
2. Develop a memory disclosure exploit.
3. Harvest code pages with the exploit.



4. Find ROP gadgets.
5. Exploit.

Advanced Defenses

What's Next?



Code injection
NX/DEP
Code-reuse attacks
ASLR
ROP on Fixed Code
ASLR w/ PIE
Canary
Memory Disclosure
(JIT ROP)
Next?

Advanced Code Randomization

Can we make code randomization resilient to JIT ROP?

Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming, ***NDSS 2015***.

Motivation of Isomeron

JIT ROP assumes that we can always leverage memory disclosure bugs to obtain code pages, and build a ROP payload at runtime.

> But can we disable code reuse attacks even after the memory is completely disclosed?

The Idea: Control Flow Randomization

Main two copies of binary images.

- Original one.
- A copy after applying fine-grained ASLR.

And randomly jump to one of the copies when there is a function call and return.

Overview of Isomeron

```
A: ...; call B
```

```
B: ...; ret
```

```
B': ...; ret
```

```
A': ...; call B'
```

Execution Diversifier

1. Identify origin.
2. Flip a coin ($r \in \{0, 1\}$), and store (SP, r) in memory.
3. Adjust return address to always point to the original return address, and jump to either B or B' depending on r.

Overview of Isomeron

```
A:  ...; call B

B:  ...; ret
```

```
B':  ...; ret

A':  ...; call B'
```

Execution Diversifier

1. Look up from the memory the decision made for the current stack ptr.
2. Adjust return address if necessary, and return.

Challenges?

- High performance overhead (19% overhead on avg.).
- Need to increase the number of copies to reduce the probability of guessing.
- How can we hide the diversifier data?

Performance vs. Security

Another Perspective: XnR

JIT ROP may not be possible if we can make code sections ***unreadable***.

You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code, ***CCS 2014***.

But Current H/W Does Not Support XnR

There's no XnR (eXecutable but not Readable) permission!

Can we emulate this with S/W?

Emulating XnR

- Set the ***present bit*** of a page false.
- Modify page fault handler to check whether the instruction is illegally reading the code.
  - Regular instruction fetch should be considered legitimate.
  - Accessing memory that contains data is legitimate.
  - But, ***accessing memory that contains code*** is illegal!

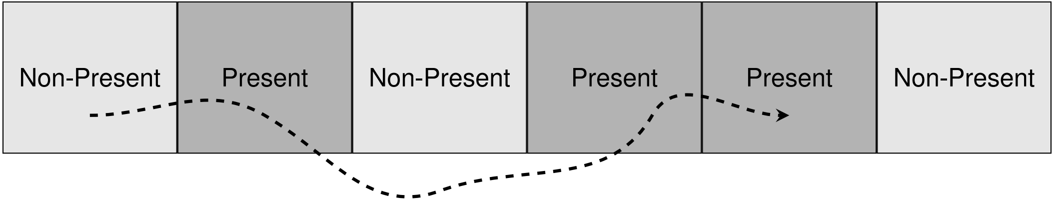
Challenges

- We should make the current page as “non-present”.
- Thus, too many page faults (performance overhead).

Can we make it faster?

Sliding-Window Approach

Control the maximum number of present pages: Most recently used N pages are "present", but all the other pages are "non-present".



Security vs. Performance Trade-Off

More secure when N is smaller, but becomes slower.

Comparison

- Isomeron
  - Make JIT ROP harder.
  - High performance overhead.
- XnR
  - Tries to fundamentally prevent memory disclosure.
  - But there is a huge gap between the ideal and the reality.
  - Memory disclosure still possible within a sliding window.

Q: Perfect XnR w/o Fine-grained ASLR?

Let's suppose there exists a way to enforce the perfect XnR policy without the performance issue, but we don't employ fine-grained ASLR. Can we say we are safe?

XnR Prevents Reading Code, But ...

- An attacker can still read stack or heap data to harvest function pointers.
- If we know a function pointer of a specific function, then we don't need to read the actual code for the function. We just get the code offline and build a ROP payload!

XnR Prevents Reading Code, But ...

- An attacker can still read stack or heap data to harvest function pointers.
- If we know a function pointer of a specific function, then we don't need to read the actual code for the function. We just get the code offline and build a ROP payload!
- This attack is so-called ***indirect JIT ROP***.

Next Research Question

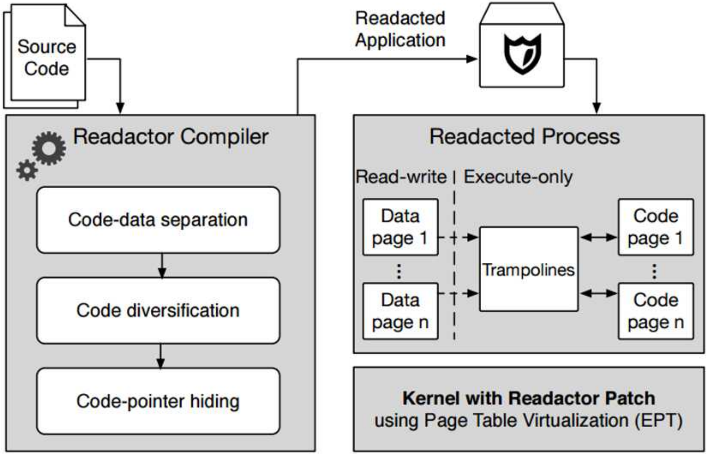
Can we mitigate both direct and indirect JIT ROP attacks?

Readactor: Practical Code Randomization Resilient to Memory Disclosure, ***Oakland 2015***.

Readactor

- Use both fine-grained ASLR and XnR.
- Implement XnR via a thin hypervisor.
  - EPT (Extended Page Table) allows the XnR permission.
- Separate code and data, and apply both fine-grained ASLR and XnR for code.
- Hide code pointers.
  - Translate jump tables into a sequence of jump statements, and put them in the code region (thus, XnR will protect them).
  - Translate return addresses (on the stack) into trampoline addresses, and put the trampolines inside the code region (XnR will protect them).

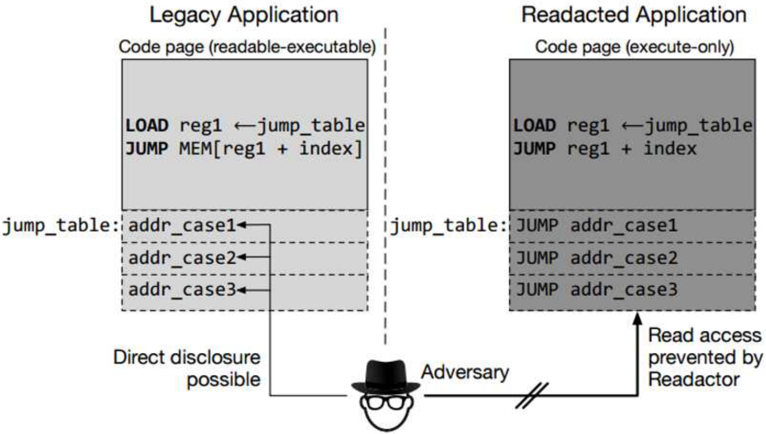
Readactor Design



[2]

[2] Image taken from Readactor, ***Oakland 2015***

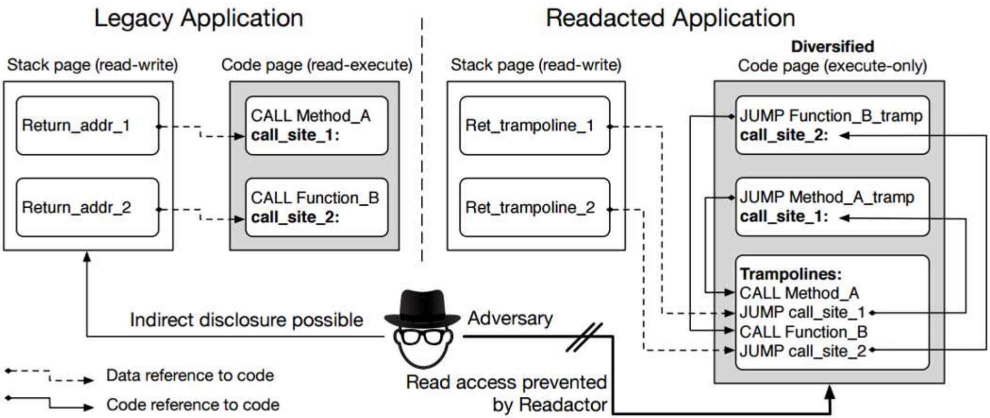
Readactor Design: Code Data Separation



[3]

[3] Image taken from Readactor, ***Oakland 2015***

Readactor Design: Code Pointer Hiding



4

[4] Image taken from Readactor, ***Oakland 2015***

Applicability?

Can we apply the Readactor defense for JavaScript engines?

- JS engines typically run JIT-compiled code at runtime.
- JIT-compiled code is typically allocated at the code cache with the RWX permission because it is frequently updated.

Readacting JIT Code Cache

- We can modify JIT compilers to separately output code and data into different pages.
- But, we still need to dynamically change the permission of code pages: Alternate RW and XnR.
  - When modifying code, suspend execution and make code pages RW.
  - When executing code, make code pages XnR.

Redactor Performance Evaluation

- Chromium Browser: avg. 4.0% slowdown.
- SPEC CPU 2006: avg. 6.4% slowdown.

Redactor Performance Evaluation

- Chromium Browser: avg. 4.0% slowdown.
- SPEC CPU 2006: avg. 6.4% slowdown.

Security vs. Performance

Q: Any Security Problems with Readactor?

When alternating RW and XnR (for JIT engines), there is a time when code pages become both readable and writable. Is this a problem?

Q: Any Security Problems with Readactor?

When alternating RW and XnR (for JIT engines), there is a time when code pages become both readable and writable. Is this a problem?

See Exploiting and Protecting Dynamic Code Generation, ***NDSS 2015***

Readings

- User-level XnR
  - XoM-Switch, ***Black Hat Asia 2018***
- XnR on ARM
  - uXOM: Efficient eXecute-Only Memory on ARM Cortex-M, ***USENIX Security 2019***

Question?

Security vs. Performance

None of the advanced defense techniques learned in this lecture is adopted in a real-world system. Why?