Hacking in Darkness: Return-oriented Programming against Secure Enclaves

Jaehyuk Lee°  Jinsoo Jang°  Yeongjin Jang★
Nohyun Kwak°  Yeseul Choi°  Changho Choi°

Taesoo Kim★  Marcus Peinado†  Brent Byunghoon Kang °
Big Idea: Cloud Computing
Big Hurdle: “Security”

62 Percent of Companies Store Sensitive Customer Data in the Public Cloud

And almost 40 percent of cloud services are commissioned without the involvement of IT, a recent survey found.

By Jeff Goldman | Posted February 21, 2017

IT leaders say it's hard to keep the cloud safe

Shadow IT causing cloud trouble by illicitly working behind the scenes

By Sharon Gaudin | Follow
Senior Writer, Computerworld | FEB 15, 2017 12:17 PM PT
SGX protects enclave from outside

Untrusted Application

Secure Enclave

Data

Code

Untrusted Operating System

CPU Package

System Administrator
Memory encryption in SGX

- MEE encrypts all contents of the enclave memory
- Decrypts using the hardware provided key
- Cold boot attacks & Snooping is impossible
Memory protection in SGX

• MMU keeps system software from accessing Enclaves
• Allows the accessibility of the enclave to its own contents
Now, can we say all software is secure?
Software vulnerabilities are prevalent

OSS-Fuzz: Five months later, and rewarding projects

Monday, May 8, 2017

Five months ago, we announced OSS-Fuzz, Google’s effort to help make open source software more secure and stable. Since then, our robot army has been working hard at fuzzing, processing 10 trillion test inputs a day. Thanks to the efforts of the open source community who have integrated a total of 47 projects, we’ve found over 1,000 bugs (264 of which are potential security vulnerabilities).

Breakdown of the types of bugs we’re finding.
Return-oriented programming (ROP) attack

```c
void vuln(char *input) {
    char dst[0x100];
    memcpy(dst, input, 0x200);
}
```

![Diagram showing return-oriented programming (ROP) attack](image)
Return-oriented programming (ROP) attack

```c
void vuln(char *input) {
    char dst[0x100];
    memcpy(dst, input, 0x200);
}
```

e.g., `system("/bin/sh")`

```
pop rdi; ret
"/bin/sh"
```
Return-oriented programming (ROP) attack

```c
void vuln(char *input) {
    char dst[0x100];
    memcpy(dst, input, 0x200);
}
```

Assumption:
Addresses of the pop gadget & function are known (e.g., reverse engineering)
Deploying an encrypted binary in SGX

• Operating System loads the enclave pages to memory.
  • Malicious OS can see the content of enclave binaries.

• Software vendor can make use of full encryption over enclave binaries.
  • Prevent the reverse engineering.
  • VC3 first showed private code can be loaded to enclaves.
Deploying an encrypted binary in SGX

- Encryption over the binary to prevent reverse engineering
ROP inside an enclave

```c
void vuln(char *input) {
    char dst[0x100];
    memcpy(dst, input, 0x200);
}
```

Code is not visible (i.e., loaded in an encrypted form)
- 0x100: ????
- 0x200: ????

For the enclave binaries
Addresses of the pop gadget & function are unknown
Threat model of Dark-ROP

• The attacker has **full control of all software of the system**
  • including the operating system and the untrusted app.

• The attacker can make the enclave program **crash multiple times**.
  • Inspecting the program behavior from the crash.

• The application is **built with a standard compiler with Intel SDK**
  • (e.g. Visual Studio for SGX, or gcc)

• Enclave application is **distributed in an encrypted format**
  • All the runtime information of the enclaves are hidden
Contribution of Dark-ROP

• We devise a new way to launch a code-reuse attack against encrypted enclave binaries
  • Finding POP gadgets to control registers in enclaves
  • Finding memcpy function to copy data from enclaves

• The Dark-ROP attack can completely disarm the security guarantees of SGX
  • Decrypting and generating the correctly sealed data.
  • Bypassing local and remote attestation.
Dark ROP: ROP in darkness

• Step 1. Finding the locations of pop gadgets
  • Pop gadget: bunch of pops followed by ret instruction.
    • pop r??; ret
    • pop r??; pop r??; ret
  • Enabling load value into the registers in enclave context

• Step 2. Locating ENCLU + pop rax (i.e., EEXIT)
  • ENCLU instruction is used to
    • Decipher pop gadgets
    • Retrieve the hardware provided key for unsealing
    • Generate the malicious report data to bypass remote attestation
• Step 3. Deciphering all pop gadgets
  • ENCLU instruction is used to decipher pop gadgets found at first step.
  • Discerning which gadget loads value to which register.
    • pop r??; ret -> pop rax; ret;

• Step 4. Locating memcpy()
  • Copying secret data from the enclaves
  • Injecting malicious data to the enclaves
### Step 1. Looking for pop gadgets

**Enclave Memory map**

<table>
<thead>
<tr>
<th>Address</th>
<th>Access Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF7500000</td>
<td>r-x</td>
</tr>
<tr>
<td>0xF752b000</td>
<td></td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td></td>
</tr>
<tr>
<td>0xF7741000</td>
<td>rw-</td>
</tr>
<tr>
<td>0xF7841000</td>
<td></td>
</tr>
<tr>
<td>0xF7842000</td>
<td>rw-</td>
</tr>
<tr>
<td>0xF7882000</td>
<td></td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td></td>
</tr>
<tr>
<td>0xF7500000</td>
<td>r-x</td>
</tr>
<tr>
<td>0xF752b000</td>
<td></td>
</tr>
<tr>
<td>0xF7741000</td>
<td>rw-</td>
</tr>
<tr>
<td>0xF7841000</td>
<td></td>
</tr>
<tr>
<td>0xF7842000</td>
<td>rw-</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td></td>
</tr>
</tbody>
</table>

Attackers have a full control over the **layout of the enclave**
Step 1. Looking for pop gadgets

Asynchronous Enclave Exit (AEX)

Enclave Context
rax = 0x00000001
rdx = 0x00000002
....

Exception Context
rax = 0x00000003
rdx = 0x00000000
....
cr2 = 0x7f181000

When Exception happens inside enclave
CPU fills synthetic state
But, we know which page incurs fault
Step 1. Looking for pop gadgets

Key idea

- Write addresses of non-executable pages on the stack

- RET to a non-executable address produces a page fault and an AEX
  - This is how we find RET instructions.

- The page incurring the fault is known (CR2 register)

- The faulting page tells us how many POPs happened before the RET

No POP gadget
Undefined behavior
Step 1. Looking for pop gadgets

One POP and a RET
Page fault at the second address

CR2 = 0xF7742000
(segfault)
Step 1. Looking for pop gadgets

- Enc code
- Enc heap

No POP gadget
Undefined behavior

CR2 = 0xF7742000
(segfault)

CR2 = 0xF7744000
(segfault)
Step 1. Looking for pop gadgets

Search entire enclave code

→ Catalog of pop gadgets (unknown args)

0xF7500002 → pop r??;
ret

0xF7500030 → pop r??;
pop r??;
ret
We still need to find the target registers

Catalog of pop gadgets (unknown args)

0xF7500002 → pop r??;
ret

0xF7500030 → pop r??;
pop r??;
pop r??;
ret
...

Catalog of pop gadgets (known args)

0xF7500002 → pop rax;
ret
0xF7500030 → pop rbx;
pop rcx;
pop rdx;
ret
...

...
Step 2. Looking for ENCLU: One opcode represents multiple functionalities

- ENCLU instruction handles all user level enclave operations.
- ENCLU behavior depends on RAX value.
- RAX = 4 -> Enclave exit.
- EEXIT does not erase enclave register values.

<table>
<thead>
<tr>
<th>Rax value</th>
<th>Leaf function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>EREPORT</td>
<td>Create a cryptographic report</td>
</tr>
<tr>
<td>0x1</td>
<td>EGETKEY</td>
<td>Retrieve a cryptographic key</td>
</tr>
<tr>
<td></td>
<td>EEXIT</td>
<td>Synchronously exit an enclave</td>
</tr>
<tr>
<td>0x4</td>
<td>EMODPE</td>
<td>Extend an access permission of EPC</td>
</tr>
</tbody>
</table>
Step 2. Looking for ENCLU instruction

• It's "required" to have a ENCLU (to exit) for proper functioning.

• Chain multiple pop gadgets we found in step 1 with a probing address.

• IF POP gadget loads $RAX = 4$ and ENCLU at probing address then EEXIT happens

Enc Stack

```
0xF7500002
0x0004
0xF7500030
0x0004
0x0004
0x0004
...
ret
```

Enc Stack

```
0xF7500002
0x0004
```

```
rax = 0x00000004
rbx = 0x00000004
rcx = 0x00000004
....
```

ENCLU ??
Step 2. Looking for ENCLU instruction

- How do we know whether eexit is invoked?
- If EEXIT happens, it will jump to address loaded in RBX register.
- If pop rax; ret & pop rbx; ret gadget was chained, enclave exits to 0x4

Enc Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF7500002</td>
<td>0x0004</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0xF7500030</td>
<td>0x0004</td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0xF751b000</td>
<td></td>
</tr>
</tbody>
</table>

Untrusted App

- ENCLU ??
- pop;ret
- pop;pop;pop;ret
- EEXIT to 0x4

Application Space

- rip = 0x00000004
- rax = 0x00000004
- rbx = 0x00000004
- rcx = 0x00000004

UNMAPPED SEGFAULT!
Step 2. Looking for pop rax; ret

- Now, locate pop rax; ret; gadget

→ EEXIT (RAX == 0x4) / AEX (RAX == 0x3)
→ Chain gadgets one by one and checks EEXIT happens

```
<table>
<thead>
<tr>
<th>Enc Stack</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF7500002</td>
<td>pop;ret</td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0xF7500020</td>
<td>pop;pop;ret</td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>0xF751b000</td>
<td>ENCLU</td>
</tr>
</tbody>
</table>

SEGFAULT
&
RAX == 0x3

AEX!!
```
Step 2. Looking for pop rax; ret

- Now, locate pop rax; ret; gadget
  → EEXIT (RAX == 0x4) / AEX (RAX == 0x3)
  → Chain gadgets one by one and checks EEXIT happens
Step 3. Deciphering pop gadgets: in search of `r?? registers`
Step 3. Deciphering pop gadgets: in search of $r??$ registers

- EEXIT (ENCLU & $rax=4$) leaves register file uncleaned
  → Scan code for all pop gadgets
  → check arguments

```
pop arg1; pop arg2;
ret
```

```
pop rax; ret
```

```
ENCLU
```
Step 3. Deciphering pop gadgets: in search of `r??` registers

- EEXIT (ENCLU & rax=4) left a register file uncleaned
  - Scan code for all pop gadgets
  - check arguments

Deciphering pop? pop? gadget

Register file

Gadget (0xF750020)

```assembly
pop arg1; pop arg2; ret
pop rax; ret
ENCLU
```

```
arg1 = 0x0001
arg2 = 0x0002
```

```
rax = 0x0004
rsi = 0x0001
rdi = 0x0002
...```
Step 4. Looking for memcpy()

- Identifying `memcpy(dst*, some valid address, 0x10)`

→ Check if “dst” contains data
Gadgets everywhere (e.g., SDK)

<table>
<thead>
<tr>
<th>Gadget</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ENCLU Gadget</em></td>
<td></td>
</tr>
<tr>
<td>do_ereport:</td>
<td></td>
</tr>
<tr>
<td>enclu</td>
<td></td>
</tr>
<tr>
<td>pop rdx</td>
<td></td>
</tr>
<tr>
<td>pop rcx</td>
<td></td>
</tr>
<tr>
<td>pop rbx</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>libsgx_trts.a</td>
</tr>
<tr>
<td>sgx_register_exception_h</td>
<td></td>
</tr>
<tr>
<td>mov rax, rbx</td>
<td></td>
</tr>
<tr>
<td>pop rbx</td>
<td></td>
</tr>
<tr>
<td>pop rbp</td>
<td></td>
</tr>
<tr>
<td>pop r12</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>libsgx_trts.a</td>
</tr>
<tr>
<td>relocate_enclave:</td>
<td></td>
</tr>
<tr>
<td>pop rsi</td>
<td></td>
</tr>
<tr>
<td>pop r15</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>pop rdi</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>libsgx_trts.a</td>
</tr>
<tr>
<td><em>Memcpy Gadget</em></td>
<td></td>
</tr>
<tr>
<td>memcpy:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>libsgx_tstdc.a</td>
</tr>
<tr>
<td><em>GPR Modification Gadget</em></td>
<td></td>
</tr>
<tr>
<td>_intel_cpu_indicator_init:</td>
<td></td>
</tr>
<tr>
<td>pop r15</td>
<td></td>
</tr>
<tr>
<td>pop r14</td>
<td></td>
</tr>
<tr>
<td>pop r13</td>
<td></td>
</tr>
<tr>
<td>pop r12</td>
<td></td>
</tr>
<tr>
<td>pop r9</td>
<td></td>
</tr>
<tr>
<td>pop r8</td>
<td></td>
</tr>
<tr>
<td>pop rbp</td>
<td></td>
</tr>
<tr>
<td>pop rsi</td>
<td></td>
</tr>
<tr>
<td>pop rdi</td>
<td></td>
</tr>
<tr>
<td>pop rbx</td>
<td></td>
</tr>
<tr>
<td>pop rcx</td>
<td></td>
</tr>
<tr>
<td>pop rdx</td>
<td></td>
</tr>
<tr>
<td>pop rax</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gadget</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ENCLU Gadget</em></td>
<td></td>
</tr>
<tr>
<td>do_ereport:</td>
<td></td>
</tr>
<tr>
<td>enclu</td>
<td></td>
</tr>
<tr>
<td>pop rax</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>sgx_trts.a</td>
<td></td>
</tr>
</tbody>
</table>

34
What can we do with all this?

- Leak sensitive information
- Permanently parasite to the enclave program
Case study 0: Dumping confidential data

- Memcpy all enclave memory contents into untrusted memory
  - i.e., memcpy(non-enclave region, enclave, size)
- Complete breakdown in enclave confidentiality
Case study 1: Compromising sealed data

Untrusted Application

Shadow_read_sealing_data()
{
  ROP_to_egetkey()
  unseal_data()
}

Secure Enclave

- RAX Gadget
- RBX Gadget
- RCX Gadget
- ENCLU Gadget
- memcpy Gadget

Sealing key

Unsealing and leaking confidential data
  i.e., EGETKEY retrieves the hardware key bound to specific enclave
Case study 2: Hijacking remote attestation

- Breaking the Integrity guarantees of SGX
  - MiTM between secure enclave and attestation server
  - Masquerading the enclave to deceive remote attestation server
Conclusion

• The first practical ROP attack on real SGX hardware
  • Exploits a memory-corruption vulnerability

• Demonstrates how the security of SGX can be disarmed.
  • Exfiltrate all memory contents from the enclave
  • Bypass the SGX attestation
  • Break the data-sealing properties

• Encourage the community
  • Explore the SGX characteristic-aware defense mechanisms
  • Develop an efficient way to reduce the TCB in the enclave.
DEMO: PoC Dark ROP

https://youtu.be/hyuZFf3QxvM

- Target binary: remote attestation example from Intel SDK
- Vulnerability: stack overflow
Q&A