Hacking in Darkness: Return-oriented Programming against Secure Enclaves

Dominik Pham

Technical University of Munich

December 2, 2018
1 Technical Background
- Intel SGX
- The ROP Attack

2 Dark-ROP Attack Design
- Finding a vulnerability
- Finding useful gadgets

3 The SGX Malware
- Extracting hidden binary from enclave
- Hijacking remote attestation as MitM

4 Mitigations
Intel SGX
Intel SGX - Security Features

- Memory encryption/isolation
- Program integrity through attestation
- Data sealing
- Deploying encrypted binary to enclave memory
<table>
<thead>
<tr>
<th>User Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLU[EENTER]</td>
<td>enter an enclave</td>
</tr>
<tr>
<td>ENCLU[EEXIT]</td>
<td>exit an enclave</td>
</tr>
<tr>
<td>ENCLU[EGETKEY]</td>
<td>create a cryptographic key</td>
</tr>
<tr>
<td>ENCLU[EREPORT]</td>
<td>create a cryptographic report</td>
</tr>
<tr>
<td>ENCLU[ERESUME]</td>
<td>re-enter an enclave</td>
</tr>
</tbody>
</table>

**Table:** The ENCLU instruction (index has to be stored in register rax).
Return Oriented Programming

- find function in with exploitable (buffer overflow) vulnerability
- exploit vulnerability to overwrite return address
  - attacker can execute any existing code (gadget)
  - attacker can chain gadgets to a ROP chain
PROBLEMS:
- Determine location of vulnerability in encrypted enclave is difficult
- Determine location of gadgets in encrypted enclave is difficult
Solution: Dark-ROP, a modified version of the ROP attack, which solves the mentioned problems
- Finding a buffer overflow vulnerability
- Finding gadgets to reuse in an encrypted enclave binary
Dark-ROP - Finding a vulnerability

- enclave program has fixed number of entry points (usually functions)
- enumerate those functions and executes them with fuzzing arguments
- on memory corruption the fall-back routine Asynchronous Enclave Exit (AEX) is triggered
  → function is candidate for vulnerability
- AEX handler stores source address of page fault in register \( cr2 \)
Requirements for enclave code:

- must contain the ENCLU instruction
- must contain ROP gadgets with at least one \textit{pop} instruction
- must contain function similar to \textit{memcpy}
Page Fault oracle:

- probe through entire executable address space of enclave memory
- after address to probe several non-executable addresses (*PF Region X*)
- if address to probe is gadget with *y* *pops*, *PF Region y* is next return address
  → will trigger AEX with address of *PF Region y* in *cr2* register
### Page Fault oracle:

#### Memory map

<table>
<thead>
<tr>
<th>Address</th>
<th>Access Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400000 - 0x408000</td>
<td>r-x</td>
</tr>
<tr>
<td>0x607000 - 0x608000</td>
<td>r-</td>
</tr>
<tr>
<td>0xF7500000 - 0xF752000 (Code)</td>
<td>r-x</td>
</tr>
<tr>
<td>0xF7741000 - 0xF7742000</td>
<td>r-w</td>
</tr>
<tr>
<td>0xF7842000 - 0xF7822000</td>
<td>r-w</td>
</tr>
<tr>
<td>0xF7883000 - 0xF7884000</td>
<td>r-w</td>
</tr>
</tbody>
</table>

#### Candidate gadget in enclave code section

- **0xF7501200**: pop rdx
- **0xF7501201**: ret

1. **Load PF_Region_1 as return address**
2. **Return to non-executable area (PF_Region_1)**

#### AEX_handler in page fault handler

```c
uint64_t PF_R[10] = {0xF7741000, 0xF7742000, 0xF7743000, 0xF7744000, ......}

AEX_handler(unsigned long CR2, pt_regs *regs) {
    // Indicate exception within enclave
    if (regs → ax == 0x03) {
        if (CR2 == 0)
            gadget = CRASH;
        else {
            int count = 0;
            foreach (uint64_t fault_addr in PF_R) {
                // verify number of pops
                if (fault_addr == CR2) {
                    number_of_pops = count;
                    break;
                }
                count++;
            }
        }
    }
    ...... 
}
```

#### Enclave Stack

<table>
<thead>
<tr>
<th>Buf[100]</th>
<th>Ret_addr (0xF7501200)</th>
<th>PF_Region_0 (0xF7741000)</th>
<th>PF_Region_1 (0xF7742000)</th>
<th>PF_Region_2 (0xF7743000)</th>
<th>PF_Region_3 (0xF7744000)</th>
<th>......</th>
</tr>
</thead>
</table>

1. **Return to candidate gadget**
2. **Load PF_Region_1 as return address**
3. **Return to non-executable area (PF_Region_1)**
4. **AEX (page fault)**
Identifying gadgets and registers oracle:

- find ENCLU instruction to call its EEXIT function
  \(\leftrightarrow\) exiting enclave with this function will not clear registers
- chain pop gadgets with value 0x4 as every argument; address to probe at the end
- EEXIT function has an address as parameter stored in rbx
  \(\rightarrow\) invoked if rax is 0x4 and address to probe is ENCLU
  \(\rightarrow\) exception thrown if value in rbx is 0x4
- repeating with distinguishable values allows us to identify the popped registers
Identifying gadgets and registers oracle:

1. pop gadget #1 (pop; ret)
2. pop gadget #2 (pop; pop; ret)
3. pop gadget #3 (pop; pop; ret)

Enclave stack:

- buf[100]
- Gadget #1
- 0x4
- Gadget #2
- 0x4
- Gadget #3
- 0x4
- ... 
- ENCLU?

Application address space:

- 0x00000004: UNMAPPED
- 0x0000000c: UNMAPPED

EEXIT_handler(pt_regs *regs, ulong error)

```c
if(error == (PF_PROT | PF_USER | PF_INSTR) && regs → ax == 0x4)
  // EEXIT happens
```

Registers:

- rax = 0x4
- rbx = 0x4
- r14 = 0x4
- r15 = 0x4
Read/Write gadget oracle:

- define source address in enclave address space *src* and a length *len*
- define destination address *dst* in untrusted memory space  
  → set *dst* and next *len* bytes to zero
- chain *pop* gadgets to put *dst*, *src* and *len* in registers *rdi*, *rsi* and *rdx*  
  with address to probe at the end
- if address to probe is *memcpy*, *dst* and next *len* bytes are non-zero
Dark-ROP - Finding `memcpy` gadgets

Read/Write gadget oracle:

```
#define BASE ((void*)0x80000000)
uint64_t zero = 0;

mmap(BASE, 0x1000, 7, MAP_ANONYMOUS | MAP_FIXED | MAP_PRIVATE, -1, 0);

ROP_to_enclave(source_addr, dest_addr, length);

if (memcmp(BASE, &zero, 0x8) != 0) {
    printf("memcpy found\n");
}
```

---

Dominik Pham (TUM)  Dark ROP against Secure Enclaves  December 2, 2018  16 / 23
We are now able to

- call any leaf function through ENCLU
- set register values which are used as parameters in leaf functions
- copy data between the untrusted and trusted address space
The SGX Malware
The SGX Malware - Extracting hidden binary from enclave

- utilizing `memcpy` gadget with
  - `src` as start of enclave’s binary
  - `dst` as address in untrusted memory space
  - `len` as size of enclave’s entire mapped space

- allows malware to mimic real enclave program
  - attacker can alter code for own purpose
Remote Attestation in SGX

Service provider

- Generate key pairs (&pub_s, &priv_s)
- Send_response (pub_s, signature)
- Verify_quote(quote)
  Compute_DH_key()
- Secure channel established!

Host operating system

ISV_APP
- Launch ISV_ENCLAVE
- Send_msg_1(pub)
- Proc_msg_2(msg2)
- REPORT
  Quote
- Send_msg_3(quote)

Quoting Enclave (QE)

ISV_ENCLAVE (trusted)
- Generate_ECDH_key_pairs (&pub, &priv)
- Copy_out_public_key(&pub)
- Compute_DH_key()
  Generate_REPORT_DATA(&pub)
  Run_EREPORT()
  Copy_out_REPORT()
- REPORT

Secure channel established!
The SGX Malware - Hijacking remote attestation as MitM

Service provider

Host operating system

ISV_ENCLAVE (trusted)

ISV_APP

TARGETINFO

REPORTDATA

REPORT

①

②

③

④

⑤

Service provider

Generate key pairs (&pub_s, &priv_s)

Send_response (pub_s, signature)

Verify_quote(quote)

The secret is shared between the remote server and the attacker

The SGX Malware (Man-in-the-Middle)

Launch ISV_ENCLAVE

Generate_ECDH_key_pairs (&pub, &priv)

Send_msg_1(pub)

Proc_msg_2(msg2) {
Compute_DH_key(

Generate_REPORTDATA(
ROP_copy_parameter(
ROP_EREPORT(
Get_Quote() 
)
Send_msg_3(quote)

REPORT

Quote

Quoting Enclave (QE)
Mitigations

- Gadget elimination
  → modify enclave code to prevent non-intended \textit{ret} instructions
  → for non-removeable gadgets: register validation after ENCLU

- Control flow integrity
  → should not use general registers for pointer
Thank you for your attention!