Mitigating Data Leakage by Protecting Memory-resident Sensitive Data

Tapti Palit, Fabian Monrose, Michalis Polychronakis

Stony Brook University, University of North Carolina – Chapel Hill, Stony Brook University
Heartbleed Attack

Real-world Impact of Heartbleed (CVE-2014-0160): The Web is Just the Start

OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160)

Original release date: April 08, 2014 | Last revised: October 05, 2016

Heartbleed bug causes major security headache

MICHAEL LIEDTKE and ANICK JESDANUN, Associated Press • April 9, 2014

Heartbleed’s Impact

35% of internet users have changed passwords or canceled accounts; 6% think their personal information was swiped

BY LEE RAINIE AND MAEVE DUGGAN
Spectre Attack

Intel Reportedly Warned of Critical Chip Security Flaws a Year Ago

CVE-ID
CVE-2017-5715 Learn more at National Vulnerability Database (NVD)
- CVSS Severity Rating - Fix Information - Vulnerable Software Versions - SCAP Mappings - CPE Information

Description
Systems with microprocessors utilizing speculative execution and indirect branch prediction may allow unauthorized disclosure of information

Intel failed to fix dangerous 'ZombieLoad' flaw affecting chips in MILLIONS of devices made by Apple, Microsoft, and Google

- Critical flaws in Intel chips continue to remain exploitable after a year
- Researchers say Intel has failed to patch them despite being aware of the issue
New Attacks Require New Defenses

• Control flow hijacking attacks are getting harder
  – Sandboxing, CFI, and other mitigations
• Data-only attacks still a possibility
• Data-only attacks do not violate the valid control flow of a program
• Spectre attacks take advantage of speculative execution side channels to leak data
Existing Solutions

• Memory Safety based approaches
  – Need to instrument every memory access instruction

• Data Flow Integrity based approaches
  – High overhead (104% of SPEC benchmarks)

• Changing data representation by XOR-ing with key
  – Easy to reverse XOR transformation
  – Difficult to keep XOR key secret

• AMD SEV, Intel SGX have different threat model
Observations About Data

- Some data is more sensitive (e.g., passwords, SSL keys)
- Data can be leaked from DRAM and caches
Overview of Our Solution

- Encrypt sensitive data in memory and caches
- Decrypt only when loading to registers
Threat Model

- Attacker can read arbitrary user-space memory
- Attacker can perform cold-boot attacks
- Attacker can not execute arbitrary code
Contributions

• Compiler-level defense against sensitive data leakage
  – Maintain confidentiality of sensitive data
    • Even when the attacker can read arbitrary memory

• Requires minimal developer intervention
  – Simple annotation-based solution

• Modest run-time overhead (max: 33%)
  – Evaluated with five popular applications
Outline

• Introduction

• Sensitive Data Domain
  – Points-To Analysis
  – Value Flow Analysis

• In-Memory Data Protection
  – AES Encryption
  – Decrypted Data Cache

• Evaluation

• Conclusion
Identifying Sensitive Data

• Compiler extension to handle annotated sensitive data

```c
void fun1 (void) {
    SENSITIVE int a;
    int b, c, d;
    int *ptr1, *ptr2;
    d = a;
    ptr1 = &a;
    ...
}
```

• Automated discovery of all sensitive memory operations
Resolving Memory Accesses via Pointers

- Sensitive data can be accessed via pointers

```c
void fun1 (void) {
    SENSITIVE int a;
    int b, c, d;
    int *ptr1, *ptr2;
    d = a;
    ptr1 = &a;
    ptr1 = &b;
    ptr2 = &b;
    ptr2 = &c;
}
```

- Use Andersen’s algorithm for points-to analysis
  - Track pointers that can point to sensitive data
Resolving Memory Accesses via Pointers

• Resolve all indirect accesses to sensitive data

```c
void fun1 (void) {
    SENSITIVE int a;
    int b, c, d;
    int *ptr1, *ptr2;
    d = a;
    ptr1 = &a;
    ptr1 = &b;
    ptr2 = &b;
    ptr2 = &c;
    ...
    printf('%d\n', *ptr1);
    ...
    ...
    int l = *ptr1;
}
```
Value Flow Analysis

• Sensitive data may propagate to other variables

```c
void fun1 (void) {
    SENSITIVE int a;
    int b, c, d;
    int *ptr1, *ptr2;
    d = a;
    ptr1 = &a;
    ...
    int l = *ptr1;
    ...
    int k = l;
    ...
}
```

• Recursively track the sources and destinations of sensitive value flows
Value Flow Analysis

• Sensitive data might propagate to other variables

```c
void fun1 (void) {
    SENSITIVE int a;
    int b, c, d;
    int *ptr1, *ptr2;
    d = a;
    ptr1 = &a;
    ...
    int l = *ptr1;
    ...
    int k = l;
    ...
}
```
Value Flow Analysis

• Sensitive data might propagate to other variables

```c
void fun1 (void) {
    SENSITIVE int a;
    int b, c, d;
    int *ptr1, *ptr2;
    d = a;
    ptr1 = &a;
    ...
    int l = *ptr1;
    ...
    int k = l;
    ...
}
```
Sensitive Equivalence Class

• Pointers may point to both sensitive and non-sensitive data

```c
void fun1 (void) {
    SENSITIVE int a;
    int b;
    fun2(&a);
    fun2(&b);
}

void fun2(int *p) {
    ...
}
```

• Build a “Sensitive Data Domain”
Sensitive Data Domain

• An “equivalence class” of sensitive data

```c
void fun1 (void) {
    SENSITIVE int a;
    int b, c;
    int *qtr = &b;
    ...
    fun2(&a);
    fun2(&b);
    ...
    c = a;
}

void fun2(int *p) {
    ...
}
```
Outline

• Introduction
• Sensitive Data Domain
  – Points-To Analysis
  – Value Flow Analysis
• In-Memory Data Protection
  – AES Encryption
  – Decrypted Data Cache
• Evaluation
• Conclusion
In-Memory Data Protection

• Store sensitive data encrypted in memory
  – Decrypt when loaded to CPU registers

• Use AES Encryption
  – Hardware-accelerated implementation using the AES-NI instructions

• Pre-generate AES round keys and store them in XMM registers
Decrypted Data Cache

- AES-NI `decrypt()` routine is expensive
- Retain decrypted content as long as possible
  - 128 bit Decrypted Data Cache

```c
... struct S {int a, int b, int c};
...
SENSITIVE struct S sobj; initialize(sobj);
...
printf("%d", sobj.a);
printf("%d", sobj.b);
printf("%d", sobj.c);
```

Diagram:
- DRAM
  - (encrypted values of `sobj`)
  - Decrypted Data Cache (XMM0)
  - 128 bit AES Block
Decrypted Data Cache

- AES-NI `decrypt()` routine is expensive
- Retain decrypted content as long as possible
  - 128 bit Decrypted Data Cache

```
... struct S {int a, int b, int c};
...
SENSITIVE struct S sobj;
initialize(sobj);
...
printf("%d", sobj.a);
printf("%d", sobj.b);
printf("%d", sobj.c);
```
Decrypted Data Cache

- **AES-NI** `decrypt()` routine is expensive
- Retain decrypted content as long as possible
  - 128 bit Decrypted Data Cache

```c
...
struct S {int a, int b, int c};
...
SENSITIVE struct S sobj;
initialize(sobj);
...
printf("%d", sobj.a);
printf("%d", sobj.b);
printf("%d", sobj.c);
```

(encrypted values of `sobj`)
Decrypted Data Cache

- **AES-NI `decrypt()` routine is expensive**
- **Retain decrypted content as long as possible**
  - 128 bit Decrypted Data Cache

```c
... struct S { int a, int b, int c }; ...
SENSITIVE struct S subj;
initialize(subj);
...
printf("%d", subj.a);
printf("%d", subj.b);
printf("%d", subj.c);
```

![Decrypted Data Cache Diagram](image)
Decrypted Data Cache

- AES-NI `decrypt()` routine is expensive
- Retain decrypted content as long as possible
  - 128 bit Decrypted Data Cache

```c
... struct S {int a, int b, int c};
...
SENSITIVE struct S subj; initialize(subj);
...
printf("%d", subj.a);
printf("%d", subj.b);
printf("%d", subj.c);
```

DRAM

(encrypted values of subj)

Decrypted Data Cache (XMM0)

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
<th>0x300</th>
<th>0xAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 bit AES Block</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation using LLVM Toolchain

- Annotated C source code
  - Clang frontend
  - LLVM Backend
- Gold linker
- Pointer analysis
- Value flow analysis
- AES transform
- Merged IR
- Static library with LTO objects
- LLVM Backend
- Protected executable
- Instruction selection
- Register allocation
- Annotated C library
- Clang frontend
- llvm-ar
Outline

• Introduction

• Sensitive Data Domain
  – Points-To Analysis
  – Value Flow Analysis

• In-Memory Data Protection
  – AES Encryption
  – Decrypted Data Cache

• Evaluation

• Conclusion
Sensitive Memory Operations

<table>
<thead>
<tr>
<th>Service</th>
<th>Code</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MbedTLS Server</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>(SSL Private Key)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lighttpd With ModAuth (Password)</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Memcached With Authentication (Password)</td>
<td>0.4</td>
<td>0.33</td>
</tr>
<tr>
<td>ssh-agent (Private Key)</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Minisign (Private Key)</td>
<td>14</td>
<td>27</td>
</tr>
</tbody>
</table>
Overhead of Memory Encryption

<table>
<thead>
<tr>
<th>Service</th>
<th>Overhead in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>MbedTLS Server (SSL Private Key)</td>
<td>13</td>
</tr>
<tr>
<td>Lighttpd With ModAuth (Password)</td>
<td>8</td>
</tr>
<tr>
<td>Memcached With Authentication (Password)</td>
<td>0</td>
</tr>
<tr>
<td>ssh-agent (Private Key)</td>
<td>4</td>
</tr>
<tr>
<td>Minisign (Private Key)</td>
<td>33</td>
</tr>
</tbody>
</table>
Conclusions

• Presented a compiler-level defense for sensitive data confidentiality

• Key benefit: instrumentation of only a fraction of memory accesses

• Modest run-time overhead with five apps
Heartbleed Attack

Malicious Field
pk_len > actual size of packet

SERVER HEAP MEMORY

64 KB Echo Response
(SSL Private Keys, Passwords)

Vulnerable OpenSSL Library

pk_len = 64KB

... SSL Private Keys, Passwords ...

Actual size of packet = 1 KB

pk_len = 64KB
**Spectre Attack**

- **Speculative Execution**: True branch executed even if `ind > ARR_LEN`

```c
void access(int ind) {
    if (ind < ARR_LEN)
        return array[ind];
    else
        return 0;
}
```

- Exfiltrate through Cache Timing Attack

SSL Private Keys, Passwords

Array of size 6

HARDWARE CACHES

array[1024]
Limitations

• **Integrity**
  – Encryption provides some integrity
  – Attacker doesn’t know encryption key, so can’t write valid values
  – But corner cases remain
    • E.g. if (isAdmin > 0) { ... do admin stuff ...}

• **Scalability**
  – Andersen’s analysis is $O(n^3)$
  – Future work to solve these problems
Precision of Static Analysis Analysis

• Precision of Static Analysis determines the size of the Sensitive Data Domain
  – Determines number of instrumented memory accesses

• Field sensitive Andersen’s points-to analysis and value-flow analysis

• More precise analysis increases analysis time
Field Sensitivity

- Treat each field of a struct independently

```
struct A
int *ptr1
int *ptr2
int *ptr3
int *ptr4
```

Field Sensitive Points-To Analysis

Field Insensitive Points-To Analysis
## Percentage of Sensitive Mem. Ops.

<table>
<thead>
<tr>
<th>Application</th>
<th>Sensitive Data</th>
<th>Code</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MbedTLS SSL Server</td>
<td>SSL Private Key</td>
<td>11%</td>
<td>15%</td>
</tr>
<tr>
<td>Lighttpd with ModAuth</td>
<td>Password</td>
<td>5%</td>
<td>11%</td>
</tr>
<tr>
<td>Memcached with Authentication</td>
<td>Password</td>
<td>0.1%</td>
<td>~0%</td>
</tr>
<tr>
<td>ssh-agent</td>
<td>Private Key</td>
<td>8%</td>
<td>3%</td>
</tr>
<tr>
<td>Minisign</td>
<td>Private Key</td>
<td>14%</td>
<td>27%</td>
</tr>
</tbody>
</table>
## Overhead Of Memory Encryption

<table>
<thead>
<tr>
<th>Application</th>
<th>Runtime (original)</th>
<th>Runtime (instrumented)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>MbedTLS SSL Server</td>
<td>110s</td>
<td>126s</td>
<td>13%</td>
</tr>
<tr>
<td>Lighttpd with ModAuth</td>
<td>37s</td>
<td>40s</td>
<td>8%</td>
</tr>
<tr>
<td>Memcached with Authentication</td>
<td>67s</td>
<td>67s</td>
<td>0%</td>
</tr>
<tr>
<td>ssh-agent</td>
<td>485s</td>
<td>469s</td>
<td>4%</td>
</tr>
<tr>
<td>Minisign</td>
<td>69s</td>
<td>54s</td>
<td>33%</td>
</tr>
</tbody>
</table>