SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs

Jaebaek Seo, Byoungyoung Lee*, Seongmin Kim, Ming-Wei Shih+, Insik Shin, Dongsu Han, Taesoo Kim+

KAIST, *Purdue, +Georgia Tech
Cloud is big thing, but security is concern
Amazon Cloud Used To Steal Financial Data

by Andrew R. Hickey  on June 6, 2011, 11:32 am EDT

Hacker Steals 58 Million User Records from Data Storage Provider

Stolen data belongs to Modern Business Solutions customers

The DrK Attack: De-randomizing Kernel ASLR (github.com)

159 points by tsgates 116 days ago  |  hide  |  past  |  web  |  30 comments
SGX is a promising solution
Intel Software Guard eXtensions (SGX)

- Provide secret region “enclave” protected from kernel and HW-based attacks
Traditional attacks (e.g., code reuse attack) are still available in SGX
Address Space Layout Randomization

• ASLR is the most popular and effective defense against code reuse attack

• ASLR is important, so Intel SGX SDK includes it but it is limited
Challenges

It is non-trivial when attacker is kernel

**P1.** Visible memory layout
**P2.** Small randomization entropy
**P3.** No runtime page permission change
Challenges

It is non-trivial when attacker is kernel

**P1.** Visible memory layout $\rightarrow$ Secure in-enclave loading

**P2.** Small randomization entropy $\rightarrow$ Fine-grained ASLR

**P3.** No runtime page permission change $\rightarrow$ Soft-DEP/SFI
P1. Visible Memory Layout

Enclave setup needs ring-0 instructions

1. Request

User space

Kernel space

Enclave

User process

Untrusted kernel
Enclave setup needs ring-0 instructions

The setup includes loading enclave program (visible to kernel)
P1. Visible Memory Layout

Enclave setup needs ring-0 instructions

The setup includes loading enclave program (visible to kernel)
P1. Visible Memory Layout

Enclave setup needs ring-0 instructions

The setup includes loading enclave program (visible to kernel)

No randomization in the view of kernel!
ASLR in Intel SGX SDK

• It only randomizes the base address of enclave that is known to kernel

• In addition, memory layout of enclave is visible to kernel

⇒ No ASLR in the view of kernel!
Secure In-enclave Loading

Enclave

Secure in-enclave loader

Code pages

Data pages

User process
Secure In-enclave Loading

- **Encrypted enclave program**
- **Secure in-enclave loader**
- **Code pages**
- **Enclave program**
- **Data pages**
- **Secure channel**

**User process**
Secure In-enclave Loading

1. Encrypted enclave program
2. Secure channel
3. Secure in-enclave loader
4. Code pages
5. Data pages
6. Enclave program
7. Runtime Data
8. SGX related data structure
9. User process
10. Secure in-enclave loading
Secure In-enclave Loading

Untrusted kernel

Hide memory layouts!!
Challenges

P1. Memory layout is visible to kernel
P2. Small physical memory (i.e., small entropy)
P3. Runtime page permission change is not supported
P2. Low Entropy

Small amount of physical memory is provided

Virtual-to-Physical mapping (i.e., paging) is managed by kernel

Brute-forcing attack
Fine-grained ASLR

Usual control flow

Sequential execution (e.g., fall-through)
Fine-grained ASLR

Usual control flow

Control flow with fine-grained ASLR

Secure in-enclave loading
Challenges

**P1.** Memory layout is visible to kernel

**P2.** Small physical memory (i.e., low entropy)

**P3.** Runtime page permission change is not supported
P3. No Runtime Permission Change

Loading and relocation

→ Write to code
P3. No Runtime Permission Change

• Current SGX does not support runtime page permission change
• We must keep some code pages writable

→ Code injection attack
## Goal of Soft Permission Enforcement

<table>
<thead>
<tr>
<th></th>
<th>Hardware-based permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out of enclave</td>
<td></td>
</tr>
<tr>
<td>Code of loader</td>
<td>RWX</td>
</tr>
<tr>
<td>Code</td>
<td>RWX</td>
</tr>
<tr>
<td>Data of loader</td>
<td>RW</td>
</tr>
<tr>
<td>Data</td>
<td>RW</td>
</tr>
<tr>
<td>Out of enclave</td>
<td></td>
</tr>
</tbody>
</table>

Virtual address space of an enclave
## Goal of Soft Permission Enforcement

<table>
<thead>
<tr>
<th></th>
<th>Hardware-based permission</th>
<th>Software+Hardware permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out of enclave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code of loader</td>
<td>RWX</td>
<td>No Permission</td>
</tr>
<tr>
<td>Code</td>
<td>RWX</td>
<td>X</td>
</tr>
<tr>
<td>Data of loader</td>
<td>RW</td>
<td>No Permission</td>
</tr>
<tr>
<td>Data</td>
<td>RW</td>
<td>RW</td>
</tr>
</tbody>
</table>

Virtual address space of an enclave
Instrumentation

Inspired by NativeClient (Oakland’ 09)

Write operation

Before:

\[
\text{mov } [rdx], \text{rax}
\]

After:

\[
\begin{align*}
\text{lea} & \quad r13, [rdx] \\
\text{sub} & \quad r13, r15 \\
\text{mov} & \quad r13d, r13d \\
\text{mov} & \quad [r15 + r13], \text{rax}
\end{align*}
\]
Implementation

• LLVM 4.0 with Clang frontend
  • 1,261 LoC

• Static linker from scratch
  • 1,043 LoC

• Secure in-enclave loader (i.e., dynamic loader) from scratch
  • 2,753 LoC
Evaluation

**Q1.** How effectively does SGX-Shield defend against code reuse attacks?

**Q2.** How much performance overhead does SGX-Shield bring for CPU-intensive workloads?

**Q3.** How much performance overhead does SGX-Shield bring for real-world application?
Effectiveness against Code Reuse Attack

• In Intel SGX SDK, attacker (i.e., kernel) knows the location of each code object without any bit to guess
  • The base address of enclave is known
  • The memory layout is completely visible

• Attacker (i.e., kernel) must guess 20-bits for a code object in SGX-Shield
Effectiveness against Code Reuse Attack

• In Intel SGX SDK, attacker (i.e., kernel) knows the location of each code object without any bit to guess
  • The base address of enclave is known
  • The memory layout is completely visible

• Attacker (i.e., kernel) must guess 20-bits for a code object in SGX-Shield

SGX-Shield statistically defends against code reuse attacks!
Small Performance Overhead in CPU intensive workload

• Test application: nbench
• Major factor of performance overhead: # of increased instructions

<table>
<thead>
<tr>
<th></th>
<th>64-bytes RU</th>
<th>32-bytes RU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only ASLR</td>
<td>1.05 %</td>
<td>7.80 %</td>
</tr>
<tr>
<td>ASLR + Soft-Enforcement</td>
<td><strong>6.89 %</strong></td>
<td>14.71 %</td>
</tr>
</tbody>
</table>
Negligible Performance Overhead in real-world workload

- Sample HTTPS server provided by mbedTLS (SSL/TLS library)
Conclusion

• Goal: designing ASLR for SGX programs
  P1. Visible memory layout to kernel
  P2. Small entropy
  P3. No runtime page permission change

• Solutions
  P1 → Secure in-enclave loading
  P2 → Fine-grained ASLR
  P3 → Software-based permission enforcement

• Conclusion
  SGX-Shield effectively defends against code reuse attacks with negligible performance overhead
Thank you!

Any question?
Backup Slides
Conflict between ASLR and Attestation

Enclave

User process

Untrusted kernel

1. Request

2. Set up enclave

3. Protected running

Enclave program

Attestation

Program owner

SGX checks integrity by measuring hash of enclave memory

Randomization changes the hash value

Conflict with attestation!