CAIN: Silently Breaking ASLR in the Cloud

Antonio Barresi, Kaveh Razavi, Mathias Payer, Thomas R. Gross

WOOT’15 - 9th USENIX Workshop on Offensive Technologies
Washington D.C., August 11 2015
Threat scenario

VM

VM

Virtual Machine Monitor

Hardware
Threat scenario

Attacker VM
Attacker has user privileges

Victim VM
Process

Virtual Machine Monitor

Hardware
Threat scenario

Attacker VM

Attacker has user privileges

Knows a memory-corruption vulnerability in Victim VM's process

Victim VM

Process

Victim OS

Virtual Machine Monitor

Hardware
Threat scenario

Attacker VM

- Attacker has user privileges
- Needs to know the exact location of code regions to construct a code-reuse attack

Victim VM

- Process
- Code

Victim OS

Virtual Machine Monitor

Hardware

Address-Space Layout Randomization (ASLR)
Threat scenario

Attacker VM

Attacker has user privileges

If the VMM uses page based same content memory deduplication

Victim VM

Process

Victim OS

Virtual Machine Monitor

Hardware
Threat scenario

Attacker VM

- Attacker has user privileges
- Infer randomized base address of libraries or executable
- 0x7f9ffaa0000

Victim VM

- Process
- Code
- Victim OS

Virtual Machine Monitor

Hardware
Threat scenario

Attacker VM

Attacker has user privileges

Perform code-reuse attack with now known code locations

0x7f9ffaa0000

Victim VM

Process

Code

Victim OS

Virtual Machine Monitor

Hardware
CAIN: Cross-VM ASL INtrospection

• New attack vector against memory deduplication

• Leaks randomized base addresses (RBAs) of
  • libraries and
  • executables
  • mapped in processes running on neighboring VMs
Memory deduplication (in VMM)
Memory deduplication (in VMM)

VM A

VM B

Physical memory

Virtual Machine Monitor

merge
Memory deduplication (in VMM)
Memory deduplication (in VMM)

VM A

VM B

Physical memory

Virtual Machine Monitor

write

Copy-on-Write
Memory deduplication (in VMM)

Virtual Machine Monitor
Memory deduplication side-channel
Memory deduplication side-channel

- Attacker can craft a page (guess)
- Wait for a certain amount of time
- Write to page and measure time
- Write time will reveal if page was shared
Memory deduplication side-channel

• Attacker can craft a page (guess)

• Wait for a certain amount of time

• Write to page and measure time

• Write time will reveal if page was shared
Memory deduplication side-channel

- Attacker can craft a page (guess)
- Wait for a certain amount of time
- Write to page and measure time
- Write time will reveal if page was shared
Memory deduplication side-channel

• Attacker can craft a page (guess)
• Wait for a certain amount of time
• Write to page and measure time
• Write time will reveal if page was shared

Page content?
How long?
Noise?
Threshold?
CAIN: Cross-VM ASL INtrospection

Sleep time detection → Filtering → Verification

20
Suitable pages to break ASLR

Page aligned \(\rightarrow\) Mostly static \(\rightarrow\) Read-only in victim VM \(\rightarrow\) Known to exist

Suitable page to break ASLR

Contains base address
Suitable page under Windows

PE File Format on Disk

- **0x5a4d**
- **DOS Header**
- **COFF Header**
- **ImageBase: 0x180000000**
- **Optional Header**
- **Section Table**
- **[Code & Data]**

PE File Format in Memory

- **0x5a4d**
- **DOS Header**
- **COFF Header**
- **ImageBase: 0x7f9ffaa00000**
- **Randomized DLL base address, 19 bits of entropy**
- **4096 bytes**
- **1st page of DLL in memory**

*Page content?*
Brute-force all addresses

<table>
<thead>
<tr>
<th>&lt;Page with RBA guess&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7f9ffa70000</td>
</tr>
<tr>
<td>0x7f9ffa80000</td>
</tr>
<tr>
<td>0x7f9ffa90000</td>
</tr>
<tr>
<td>0x7f9ffaa0000</td>
</tr>
<tr>
<td>0x7f9ffab0000</td>
</tr>
<tr>
<td>0x7f9ffac0000</td>
</tr>
<tr>
<td>0x7f9ffad0000</td>
</tr>
</tbody>
</table>

\[\text{detect\_shared\_pages()}\]

0x7f9ffaa0000
Brute-force all addresses

Brute-force all possible RBAs
Wait for how long?

• Depends on the memory deduplication implementation

• Varies depending on amount of memory used

• Attacker trade-off
  • Waiting too little obstructs the attack
  • Waiting too long increases attack time
Sleep-time detection

- Create random buffer
- Copy every second page of 1st half to the 2nd half
- Start with test time $t_{\text{start}}$
  - Detect merged pages
  - Iterate and increase test time until detection rate is near 100%

$N = 10,000$, $t_{\text{start}} = 10\text{min}$
Detect merged pages

Non-shared

Merged

Non-shared
Detect merged pages

$\text{Noise? Threshold?}$

$t_1$ → Non-shared → Merged → Non-shared
Detect merged pages

\[ t_1 \rightarrow \text{Non-shared} \]
\[ t_2 \rightarrow \text{Merged} \]
\[ \text{Non-shared} \]

Noise? Threshold?
Detect merged pages

Measure write time with `rdtsc` (Read Time Stamp Counter)
Detect merged pages

Measure write

time with rdtsc

(Read Time Stamp Counter)

t_1

Non-shared

29

Merged

2667

Non-shared

34

t_2

T_{1,3} < M = 1000

t_3

\[ t_2 > 2 \cdot \frac{(t_1 + t_3)}{2} \]

\[ t_1 < t_3, (t_3 - t_1) < \frac{t_3}{3} \]
Detect merged pages

These heuristics worked for different HW configurations

\[ t_2 > 2 \times (t_1) \]

\[ T_{1,3} < M = 1000 \]

\[ t_1 < t_3, (t_3 - t_1) < t_3/3 \]
Handling noise

• Noise can affect write time
  • High write time for non-shared page
  • Miss shared page because of increased write time of adjacent pages

• Perform several rounds of detection
  • Noise will cancel out
  • Eliminate candidates
Handling noise

Round 1

<table>
<thead>
<tr>
<th>G1</th>
</tr>
</thead>
<tbody>
<tr>
<td>G2</td>
</tr>
<tr>
<td>G3</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Gn</td>
</tr>
</tbody>
</table>
Handling noise

Round 1

<table>
<thead>
<tr>
<th>G1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G2</td>
<td></td>
</tr>
<tr>
<td>G3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Gn</td>
<td></td>
</tr>
</tbody>
</table>

---

```
detect()
```

<table>
<thead>
<tr>
<th>G22</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G37</td>
<td></td>
</tr>
<tr>
<td>G98</td>
<td></td>
</tr>
</tbody>
</table>

False positive
False positive
Noise? Threshold?
Handling noise

Round 1

G1
G2
G3
...
Gn

detect() →

G22
G37
G98

Round 2

G22
G37
G98

detect() →

G22
G37

False positive

False positive

False positive

Noise? Threshold?
Handling noise

Round 1

G1
G2
G3
...
Gn

detect() →

G22
G37
G98

Round 2

G22
G37
G98

detect() →

G22
G37

Round 3

G22
G37

detect() →

G37

Noise? Threshold?
False positive
False positive
False positive
Filtering

- As little memory per candidate as possible
- Once at the beginning to eliminate as many candidates as possible
Verification

• Verify remaining potential candidates

• Use more memory per candidate

• Eliminate remaining false positives
CAIN attack phases

Sleep time detection
Goal: detect a wait time \( t \) required to merge pages

Filtering
Goal: reduce number of candidates with as little memory as possible
1 round

Verification
Goal: verify remaining candidates and remove false positives
\( N \) rounds
Evaluation
Evaluation setup

- Dual CPU Blade Server
  - 2 x AMD Opteron 6272 CPUs with 16 cores each
  - 32GB of RAM
  - VMM: KVM on Ubuntu Server 14.04.2 LTS x86_64
  - Linux Kernel 3.16.0

- 1 attacker VM with Ubuntu Linux 14.04
- 7 victim VMs with Windows Server 2012 (6.2.9200 Build 9200)
  - 4 vCPUs, 4 GB per VM
ASLR in Windows x64

- High Entropy ASLR
- 33 bits for stacks
- 24 bits for heaps
- 17 bits for executables
- 19 bits for DLLS

System-wide at boot-time
Attacking a single Windows VM

![Graph showing ASLR entropy over attack time]

- **ASLR entropy (bhits)**
- **Time t = 0**

- **200 sleep_millisecs**
Attacking a single Windows VM

19 bits of entropy
524,288 potential base addresses

time $t = 0$
Attacking a single Windows VM

19 bits of entropy
524,288 potential base addresses

Sleep-time detection = 96 min per round

attack time (min)

ASLR entropy (bits)

time t = 0
Attacking a single Windows VM

After first round 6,550 candidates remain, entropy = 12.68

time t = 96
Attacking a single Windows VM

After the second round, 5 candidates remain, entropy = 2.32

\[
\text{time } t = 192
\]
Attacking a single Windows VM

Down to 1 candidate, the actual base address

time t = 288
Attacking a single Windows VM
and various merge times
Attacking a Windows VM under load

Victim VM runs IIS
webserver, load generated with a
separate physical
machine and AB
(apache benchmark)
sleep_millisecs = 20
Attacking multiple Windows VMs

![Graph showing ASLR entropy over attack time for different numbers of victims.](image)

- 1 victim
- 2 victims
- 3 victims
- 4 victims
- 5 victims
- 6 victims

sleep_millisecs = 20
Post-CAIN exploitation

• De-randomized base address can now be used in code-reuse attack

• Against a single victim, use the inferred value

• Against multiple victims, try all base addresses
  • Or link base addresses to specific VMs or processes
  • E.g., through Import Address Table (Windows) or .got.plt (Linux), or any other suitable page
Mitigations

- VMM layer: Deactivation of memory deduplication
- System layer: Attack detection
- ASLR layer: Increase ASLR entropy
- Process layer: More entropy in sensitive memory pages
Conclusion

• CAIN effectively leaks RBAs through VMMs with same content page-based memory deduplication
  • Impact: ASLR no longer effective as defense!
  • Attacking Windows < 5 hrs
  • Higher entropy ASLR > days (e.g., Linux)

• Reliable and automated PoC exploit
• Real-world cloud providers affected
• Vendors notified on June 4, 2015 (CVE-2015-2877)
Thanks!

antonio.barresi@inf.ethz.ch