HEISENBYTE:
Thwarting Memory Disclosure Attacks using Destructive Code Reads

Adrian Tang
Simha Sethumadhavan
Salvatore Stolfo
Key Idea: Destructive Code Reads

**Problem**

Executable memory can be read

Memory disclosure bugs

Dynamic code reuse attacks

**Our Solution**

Make executable memory indeterminate after it has been read
Our Inspiration

Observer Effect:
“The act of observing a system inevitably changes the state of the system.”

HEISENBYTE’s destructive code reads:
“Reading executable memory changes the executable state of the read memory.”

Executing memory after reading it yields unpredictable behavior
# HEISENBYTE in a Slide

<table>
<thead>
<tr>
<th>Dynamic Code Reuse Attack</th>
<th>Prior Defenses</th>
<th>Our Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory disclosure + Scan memory at runtime for gadgets + Chain gadgets to generate shellcode + Redirect control flow</td>
<td>Memory disclosure + Chain gadgets to generate shellcode + Redirect control flow</td>
<td>Memory disclosure + Scan memory at runtime for gadgets + Chain gadgets to generate shellcode + Redirect control flow</td>
</tr>
</tbody>
</table>

- **Tolerates discovery of code reuse gadgets, but prevents them from being used as intended**
- Extends the benefits of execute-only memory to closed source COTS binaries, especially on Windows
Why defend at Step 3?

Extends the benefits of execute-only memory to closed source COTS binaries, especially on Windows

1) Addresses the problem of incomplete separation of data from code in (Windows) COTS binaries

2) Protects transparently legacy programs that mix data and code in executable JIT dynamic code
Outline

• Destructive Code Reads

• System Implementation

• Evaluation

• Future Work
Destructive Code Reads

Detecting read operations into executable memory
Destructive Code Reads

Detecting read operations into executable memory

Mark this memory page as execute-only
Destructive Code Reads

Detecting read operations into executable memory
Destructive Code Reads

Detecting read operations into executable memory

Instruction is fetched into CPU

Instruction Pipeline

MMU

RAM

EIP: 0x202
jmp 0x100[eax*4]
Destructive Code Reads

Detecting read operations into executable memory

Virtual → Physical
0x100 → 0x100

Instruction Pipeline

CPU

MMU

RAM

EIP: 0x202
jmp 0x100[eax*4]
Destructive Code Reads

Detecting read operations into executable memory

Virtual  Physical
0x100 → 0x100

**CPU tries to read from execute-only memory**

**EIP: 0x202**
jmp 0x100[eax*4]

Read into executable memory detected
Destructive Code Reads

“Destroying” the executable byte that is read

Virtual | Physical
0x100 → 0x100

EIP: 0x202
jmp 0x100[eax*4]
Destructive Code Reads

“Destroying” the executable byte that is read

Virtual: 0x100 → Physical: 0x100

Duplicate original executable memory page

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Hexadecimal</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>94 C3 00 00</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>00 30 00 00</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x200</td>
<td>33 C0</td>
<td></td>
</tr>
<tr>
<td>0x202</td>
<td>FF 24 85</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00 01 00 00</td>
<td></td>
</tr>
</tbody>
</table>

EIP: 0x202
jmp 0x100[eax*4]
Destructive Code Reads

“Destroying” the executable byte that is read

Virtual Physical
0x100 → 0x100

```
Instruction Pipeline

CPU

MMU

RAM

0x100: 00 30 00 00
0x104: ...
0x200: 33 C0
0x202: FF 24 85
0x202: 00 01 00 00
0x1100: 94 C3 00 00
0x1104: 00 30 00 00
...:
0x1200: 33 C0
0x1202: FF 24 85
0x1202: 00 01 00 00

EIP: 0x202
jmp 0x100[eax*4]
```

Destroy the original executable byte
Destructive Code Reads

“Destroying” the executable byte that is read

Virtual to Physical
0x100 → 0x1100
Destructive Code Reads

“Destroying” the executable byte that is read

Virtual  Physical
0x100  →  0x1100

Service the original read operation
Destructive Code Reads

“Destroying” the executable byte that is read

```
Virtual       Physical
0x100        →  0x100
```

```
Instruction Pipeline
```

```
<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x100</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>FF C3 00 00</td>
</tr>
<tr>
<td>0x104</td>
<td>00 30 00 00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x200</td>
<td>33 C0</td>
</tr>
<tr>
<td>0x202</td>
<td>FF 24 85</td>
</tr>
<tr>
<td>0x206</td>
<td>00 01 00 00</td>
</tr>
<tr>
<td>0x1100</td>
<td>94 C3 00 00</td>
</tr>
<tr>
<td>0x1104</td>
<td>00 30 00 00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x1200</td>
<td>33 C0</td>
</tr>
<tr>
<td>0x1202</td>
<td>FF 24 85</td>
</tr>
<tr>
<td>0x1206</td>
<td>00 01 00 00</td>
</tr>
</tbody>
</table>

EIP: 0x202
jmp 0x100[eax*4]

```

Restore MMU translation
Destructive Code Reads

Stopping a dynamic code reuse attack

Assume memory at 0x100 was disclosed as part of an attack
Destructive Code Reads

Stopping a dynamic code reuse attack

Attacker assumes that he found a stack pivot gadget
Destructive Code Reads

Stopping a dynamic code reuse attack

The desired gadget was not executed
Outline

• Destructive Code Reads
  • System Implementation
• Evaluation
• Future Work
System Implementation

1. Offline Static Binary Rewriting
2. Initialization of Executable Memory
3. Active Monitoring Mode
System Implementation

1. Offline Static Binary Rewriting
   - Static program binaries
   - Binary rewriting
   - Rewritten program binaries

2. Configure Execute-only Memory
   - Static rewritten binaries
   - New process loaded
   - Identify executable memory pages
   - Queue
   - Mark pages as execute-only using EPT

3. Active Monitoring Mode
   - #EPT read violation
   - Destructive code read operation
Key Requirements for Destructive Code Reads

“When” to mediate?
Detect read operations into executable memory

“How” to mediate?
Maintain separate code/data views for same (virtual) memory address

Hardware-Assisted Nested Paging is a key enabler
Hardware-Assisted Nested Paging

Hardware feature to improve virtualization performance: Translate guest to host addresses in hardware

Different implementations:
  - Intel EPT*
  - AMD RVI

* EPT: Extended Page Tables
  RVI: Rapid Virtualization Indexing
When to Mediate

(1) Efficient detection of reads into executable memory

**Problem:** OS native paging cannot mark memory as execute-only

![Diagram showing virtual and physical memory addressing with page tables and pageFaults](image-url)
When to Mediate

(1) Efficient detection of reads into executable memory

**Problem:** OS native paging cannot mark memory as execute-only

**Solution:** Virtualize the host and use Intel EPT to mark execute-only

Reads into this page will trap into hypervisor
How to Mediate

(2) Efficient maintenance of separate code/data views

**Goal:** Induce different program behavior at the same virtual address depending on read or execute operation

**Solution:** Manipulate EPT to redirect memory translation at runtime

![Diagram showing memory translation and access permissions between guest and host machines.](image-url)
Architecture (Para-virtualized)

Offline
- Rewritten binary
- Relocated data
- .reloc

Live Target System
- Loaded application
- Guest mode component
- Host mode component
- Heisenbyte

Guest User
- Configure execute-only mem
- When and how to mediate

Guest Kernel
- Destroy code when read

Host
- Guest page tables
- EPT

Loaded application

11/21
Architecture (Para-virtualized)

- Offline
  - Rewritten binary
  - Relocated data
  - .reloc

- Live Target System
  - Loaded application
  - Guest mode component
  - Host mode component
  - EPT

- Guest User
  - Guest Kernel

- Host

- Live Target System components:
  - Identify executable mem
  - Induce COW

11/21
Tracking Runtime Executable Memory

How to identify executable memory we want to protect?

(1) Static program binaries
   - Windows OS-provided runtime callbacks for
     - New/exiting processes
     - Loaded libraries

(2) Dynamic JIT code
   - Inline hooking of Windows memory management APIs
   - Perform hypercalls to hypervisor when
     - Exec buffer → Non-exec
     - Non-exec buffer → Exec
     - Exec buffer → Freed

[More in paper ...] Optimizations and Windows-specific implementation details
Tracking Runtime Executable Memory

Challenges

**Challenge 1:** Shared physical memory pages across processes

**Solution:** Induce Copy-On-Write (COW) on pages with 1-byte identity write operation to each page

**Challenge 2:** Demand paging – pages could be paged out

**Solution:** Make pages resident in physical memory using `MmProbeAndLockPages()` kernel API
Some Caveats to HEISENBYTE

• Cannot handle code that reads/writes to *itself*
  • Eg. Self-modifying code

• Cannot mitigate attacks that reveal contents of memory *without directly reading* executable memory
  • Eg. Fault-based side-channel attacks (Blind-ROP)

• Need support for *fine-grained* ASLR
  • Eg. Instruction-level in-place code randomization

• *One-byte* code “destruction” regardless of operand size of read operation
Outline

• Destructive Code Reads
• System Implementation
• Evaluation
• Future Work
“Destructive code reads” overhead depends on how imperfect the separation of data from code in executable sections
Virtualization avg overhead: ~1.8%
Destructive code reads avg overhead: ~16.5%
Evaluation – Memory Overhead

Peak RSS memory avg overhead: ~0.8%
HEISENBYTE corrupts code with debug trap code $0xCC$

Crafted dynamic code reuse exploits and monitor for invoked debug trap

(1) Dynamic code
   • Self-injected bug in toy program that mimics the creation of a JIT code buffer

(2) Static code
   • CVE-2013-2551: Internet Explorer Bug

Exploits on both static programs and dynamic JIT code triggered debug traps
Evaluation – Demo on Win8 / IE10

Suspicious activity detected!

PROGRAM: iexplore.exe (pid = 1200)
Attempting to execute code that was previously read.
Dumping code dump now for analysis.

OK
Outline

- Destructive Code Reads
- System Implementation
- Evaluation
- Future Work
Future Work

• Improve code/data separation task in disassembly for Windows COTS binaries
  • Record read operations into executable memory to guide disassembly and binary rewriting

• Lower overhead of destructive code reads
  • Use new virtualization-based hardware features in Haswell+ processors (Eg. New #VE exception)

• Explore value of destructive data reads
Conclusions

**Key Idea:** Make exec. mem. indeterminate after it has been read

- New security concept: “Destructive code reads”
- One application: Mitigate memory disclosure attacks
- Heisenbyte is a practical solution
  - Works with imperfect disassembly on COTS binaries
  - No instrumentation on the binaries
  - JIT code works too

Thank you!

Adrian Tang • @0x0atang