

CONTROL FLOW INTEGRITY USING HARDWARE COUNTERS

Zero Exploit Tolerance

By Jamie Butler and Cody Pierce



Agenda

- Who we are
- The exploit problem
- Modern software vulnerabilities
- Hardware-assisted exploit prevention
- Prior research
- Leveraging hardware capabilities
- Building a CFI policy
- Efficacy
- Conclusion
- Questions

Who we are

Jamie Butler

- 20+ years in computer security
- Windows internals/kernel guy
- Primary focused on endpoint security
- Black Hat Review Board
- Co-author of *Rootkits: Subverting the Windows Kernel*
- CTO of Endgame



Who we are

Cody Pierce

- 15 years in computer security
- Led vulnerability research teams
- Technical Director of Research and Strategy at Endgame
- Frequent speaker at Black Hat





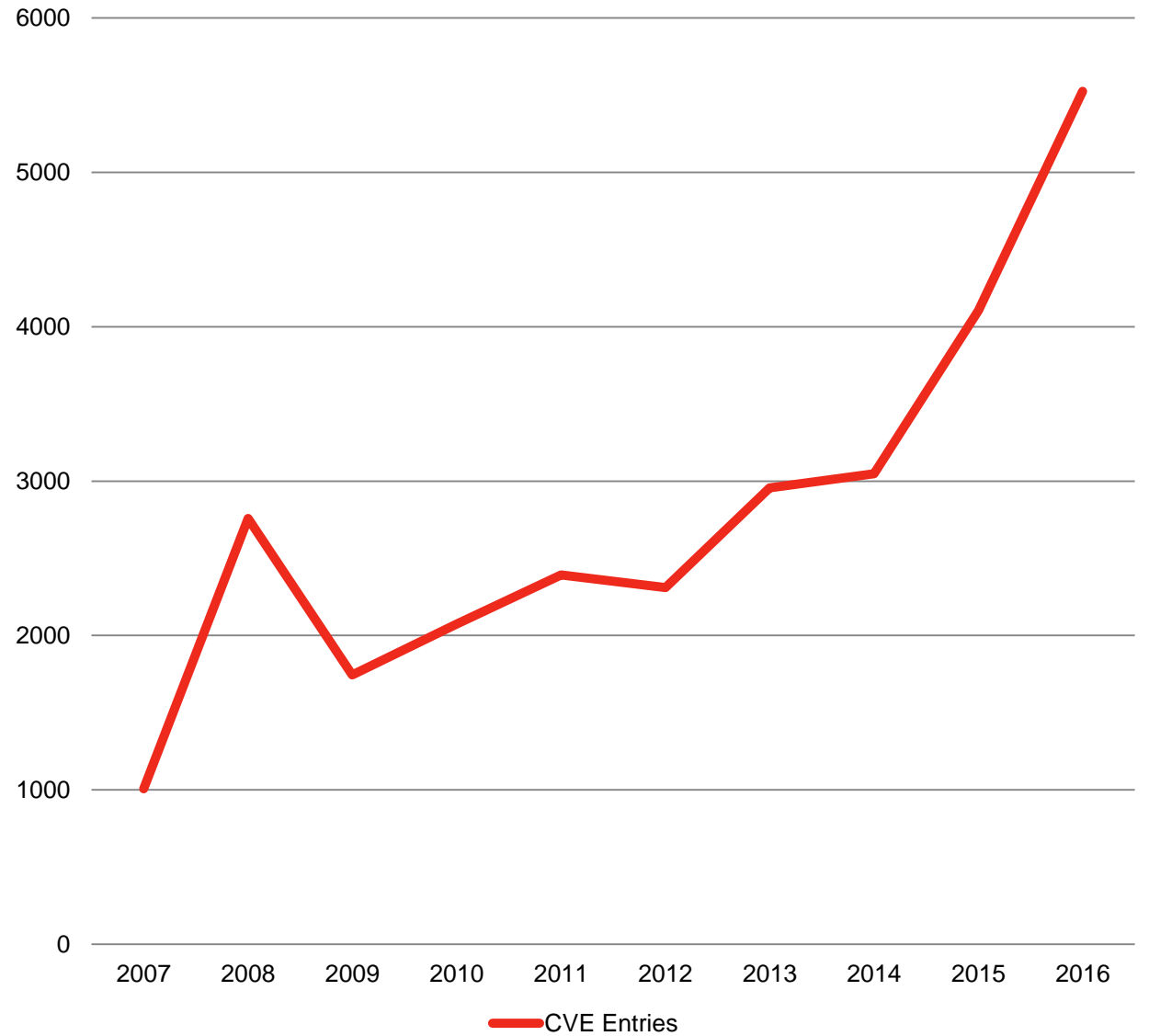
The exploit problem

- Vulnerabilities covering 2007 - 2017
- NVD CVE data set
- CVSS Score Medium+
- Counted vendors have a minimum of 5 CVE per year
- Category grouping using CWE (Common Weakness Enumeration)
- 27,922 Qualifying CVE Entries

The exploit problem

CVE Entries 2007 - 2017

Total CVEs Over Time



Why it matters

- WannaCry
 - Infected over 300,000 computers
 - Affected over 150 countries
- NotPetya
 - Targeted MeDoc
 - Maersk
 - Over \$300M
 - Halt operations at 76 ports





Modern Software Vulnerabilities



Exploit targets

- Server side vulnerabilities - require no user interaction but are heavily protected and monitored
 - Web frameworks
 - Web servers
 - Web app vulnerabilities
- Client side vulnerabilities - require limited user interaction but can bypass perimeter protections
 - Email clients
 - Web browsers
 - Document readers



Intro to commonly exploited bug classes

- A “bug class” is a taxonomy of vulnerability types, and how they manifest in code
- We use these to classify data sets, analyze trends, and develop protections
- New bug classes are discovered regularly, especially when new technologies emerge or gain popularity
- Successful exploitation of these bug classes leads to control over the target program and by proxy the host system

A person is seen from behind, looking at a large screen in a dark room. The screen displays a world map. The person is wearing a light-colored shirt. The room is dimly lit, with some lights visible in the background.

Use-after-free

- Program allocates Object A
- Program frees Object A
- Program still has a reference to Object A and accesses it despite being freed



Use-after-free code example

```
#include <new>

class ObjectA {
    void foo();
};

void uaf() {
    A *a = new ObjectA;
    // ...
    delete a;
    // ...
    a->foo();
}
```

A person is seen from behind, looking at a large screen displaying a world map. The room is dark, with some lights visible in the background, suggesting a control room or a server room.

Heap overflow

- Function A allocates Buffer A of 16 bytes
- Function A copies user data into Buffer A without checking its size
- Memory following Buffer A is overwritten with user data

Heap overflow code example

```
#define MAX_USER_DATA 16

void heapOverflow(char* userData,
size_t userDataSize)
{
    char* newUserData;

    newUserData =
(char*)malloc(MAX_USER_DATA);
    memcpy(newUserData, userData,
userDataSize);
}
```

A person is seen from behind, looking at a large screen displaying a world map. The room is dark, with some lights visible in the background.

Out-of-bounds array access

- Function A allocates a statically sized table consisting of a max size
- User can specify arbitrary list of elements and the offset into table without checking bounds
- Statically sized table can be accessed out-of-bounds by supplying a large offset into the table
- `table[index] = value`
- If `index > 16` the value is written to memory outside of the original allocation limits



Out-of-bounds array access code example

```
#define MAX_TABLE_SIZE 16

int fakeTable[MAX_TABLE_SIZE];

void oobArray(int index, int value)
{
    fakeTable[index] = value;
}
```




Type confusion

- Object A contains 1 property
- Object B contains 1 function pointer foo
- Program allocates Object A and stores user supplied data in property 1
- Program casts Object A to Object B and executes Object B foo
- Due to type confusion B->foo points to A->data and executes user data as a function pointer

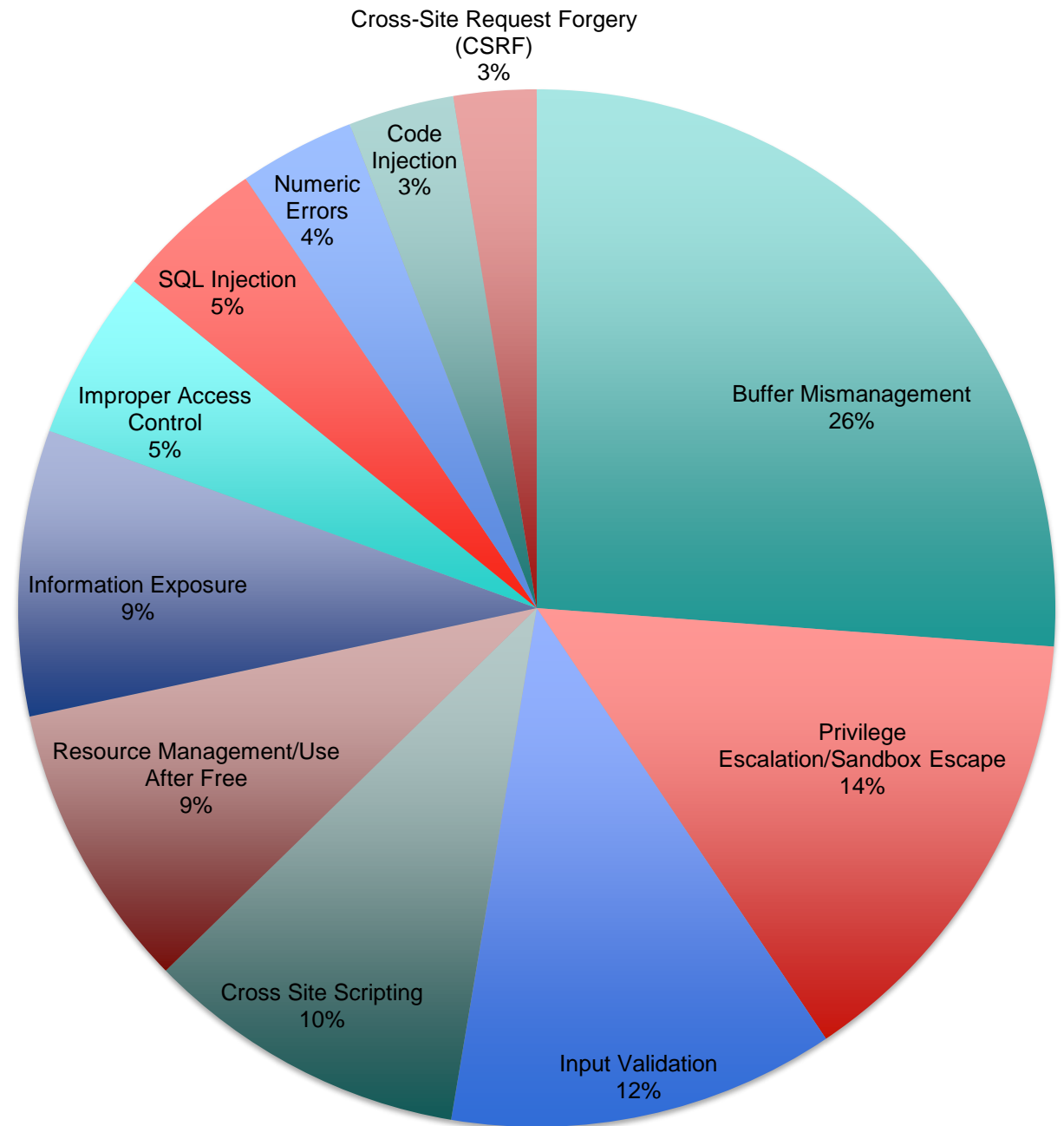
Type confusion code example

```
class ClassA {
    int userData;
};

class ClassB : ClassA {
    void foo();
}

Void typeConfusion(int exploitAddress)
{
    ClassA *ObjectA = new ClassA();
    ClassB *ObjectB;
    // ...
    ObjectA->userData = exploitAddress;
    // ...
    ObjectB =
static_cast<ClassA*>(ObjectA);
    ObjectB->foo();
}
```

CWE distribution 2007 - 2017



A person in a dark suit is seen from behind, standing in a server room. They are looking at a large screen that displays a world map. The room is dimly lit, with rows of server racks visible in the background.

Operating system mitigations

- OS vendors and processor manufacturers add new and novel protections for preventing exploitation of these vulnerabilities
- The goal is to increase the difficulty while maintaining software compatibility and performance
- ASLR, DEP, Sealed Classes, Memory Randomization, Stack Canaries to name a few

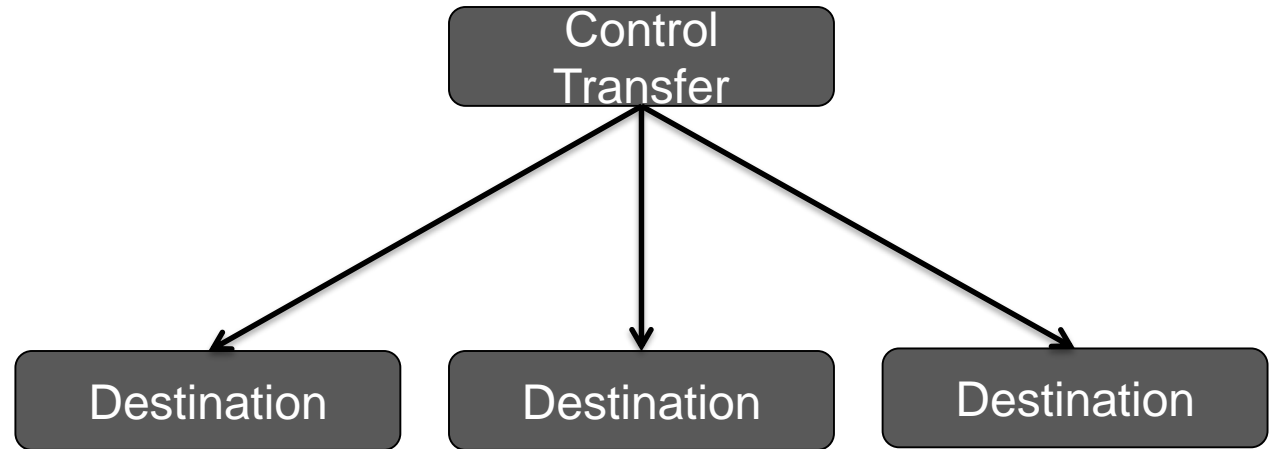


Control flow integrity (CFI)

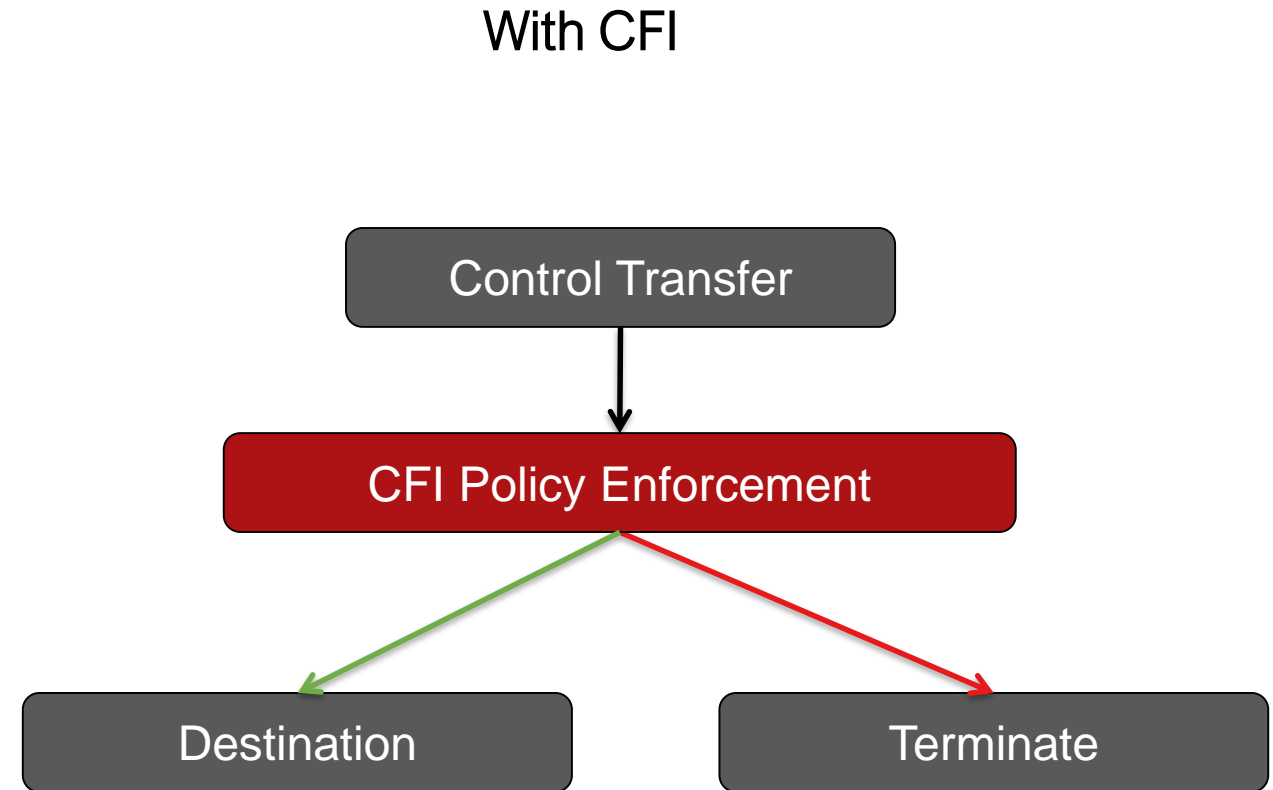
- CFI is one of the newer protections included in some compilers, most notably Microsoft Windows 10 under the name “Control Flow Guard”
- CFI is based on a policy allowing proper program execution, preventing exploits from hijacking program execution
- In all implementations thus far, recompilation is required to utilize these protections

CFI policy in practice

Without CFI



CFI policy in practice





Exploit takeaways

- Nearly half of severe vulnerabilities fall into one of these bug classes
- Attackers will exploit these types of vulnerabilities to hijack control flow and compromise the target system
- Vendors consistently add new ways to prevent successful exploitation of software vulnerabilities
- Control flow integrity is a leading way of preventing exploitation of common software, but requires recompilation and redistribution of applications



Hardware-assisted Exploit Prevention



**Prior research into
hardware-assisted
exploit prevention**

“kBouncer: Efficient and Transparent ROP Mitigation”, Pappas, 2012

“Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters”, Yuan et al., 2011

“CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters”, Xia et al., 2012

“Transparent ROP Detection using CPU Performance Counters”, Li & Crouse, 2014

A person in a dark shirt is seen from behind, looking at a world map in a server room. The room is filled with server racks and the lighting is dim, with some lights visible on the racks.

Functional requirements

- Implementation must work on Intel and AMD
- Implementation must work on 32 and 64bit Operating Systems
- CFI policies must be applied without software recompilation or access to source code
- No offline preprocessing of the program in any way
- Performance overhead of the solution should be minimal



Resilience requirements

- Additional code will not be added to the running program in the form of “hooks” or validation logic
- It must work in the Kernel
- The system must be able to detect and prevent an exploit in real-time




Leveraging Hardware Capabilities



Performance monitoring unit (PMU)

- The PMU is dedicated silicon in a processor architecture that provides platform developers with hardware level counters for deep introspection into code execution
- Dozens of PMU facilities are available include CPU cycles, Memory performance, UI, and Context switching
- Many performance tools in popular IDEs utilize these capabilities to analyze a running program
- The PMU can interrupt the processor to inspect and retrieve data

A person in a dark suit is seen from behind, looking at a world map in a server room. The room is dimly lit with rows of server racks in the background.

Branch prediction unit (BPU)

- To optimize processor cycles a BPU is used to pre-stage instructions in the instruction pipeline
- This pipeline is responsible for the ordering of execution on a running processor
- By leveraging the BPU to predict the next instructions a program significant performance improvements can be gained especially in multi-core architectures
- Unfortunately this piece of the silicon is fairly undocumented and often considered intellectual property



Indirect call mispredictions

- One feature of the PMU is to interrupt the system when programmable BPU events occur
- One such event available is to interrupt on indirect branch mispredictions
- Indirect branches occur most often when virtual functions are being executed
- As previously described in the bug-class overview, attackers are likely to hijack code execution, often through an indirect call



Interrupt vector table

- To manage the interrupts and interrupt handlers in a process a map of 255 locations is kept
- This map is tied to the interrupt vector value, defined by the processor manufacturer, and contains the function to call when that interrupt is serviced
- To effectively implement hardware based CFI we have to set up the IVT for PMI to be handled by our driver



Hardware takeaways

- Modern architectures include performance monitoring facilities that enable low-level interrupt of the processor
- The branch prediction unit is responsible for optimizing instruction pipelines and can be used by the PMU to interrupt on mispredictions
- Indirect calls are a popular target of exploits and will generate a misprediction event before control flow is hijacked



Building a CFI Policy



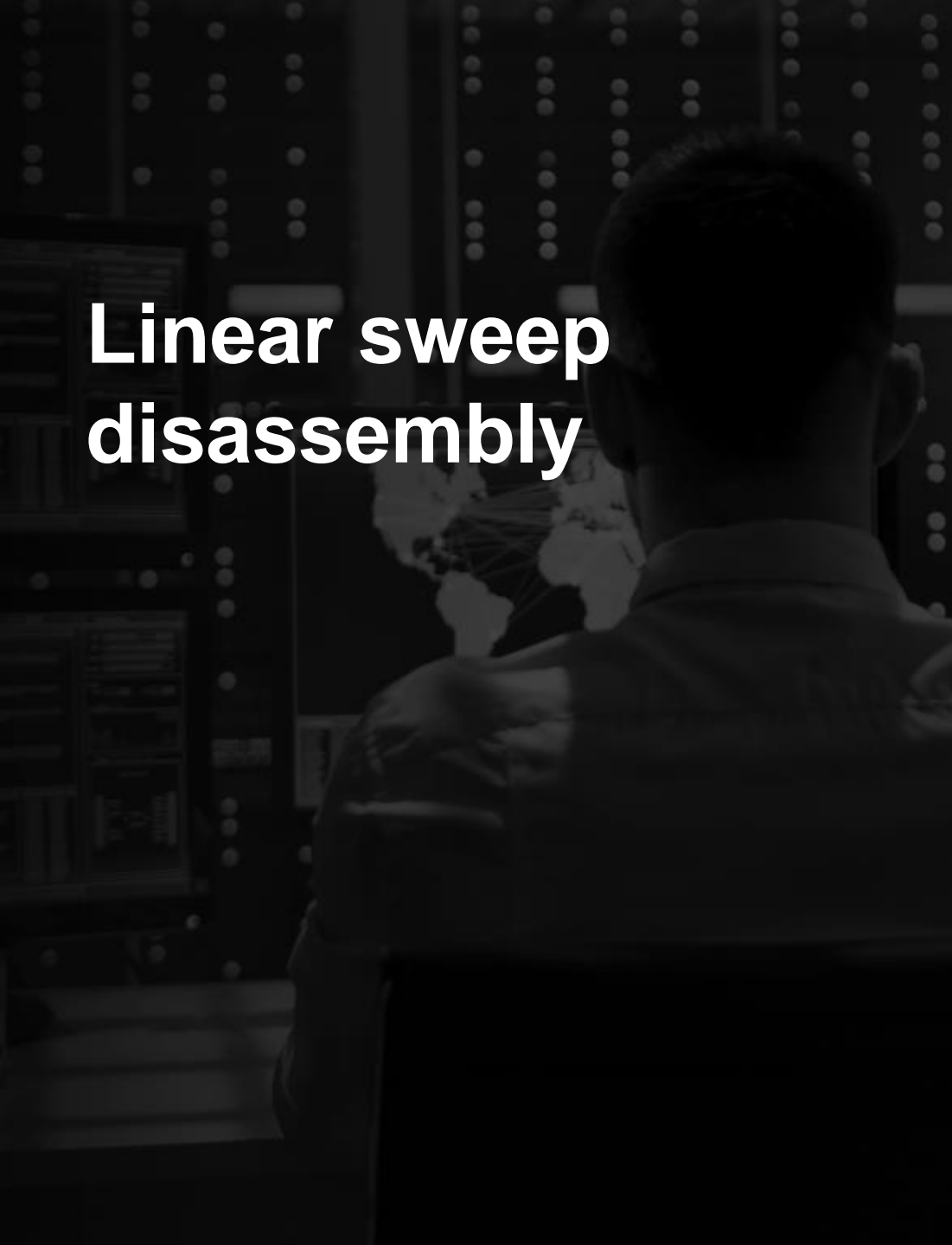
Dynamic program analysis

- Each executable image that loads into a process is an opportunity for an attacker to exploit
- Comprehensive program analysis is required to define every possible legitimate function so that a CFI policy can enforce it
- We can do this at runtime using kernel capabilities



Kernel image load callbacks

- Every Kernel has a mechanism for alerting drivers to new executable images loading into memory
- This callback is given the opportunity to process the image
- To build a valid set of control flow targets we need to identify every possible function in an image

A person in a dark suit is seen from behind, looking at a world map displayed on a wall in a server room. The room is dimly lit with rows of server racks visible in the background.

Linear sweep disassembly

- Linear sweep disassembly is the process of sequentially processing each byte in a file and disassembling them to platform instructions
- This is the most efficient way but has drawbacks vs other approaches
- Different compilers emit various assembly instructions we can build into function heuristics
- We have discovered several of these heuristics that allow us to build a set of legitimate functions in any executable image



Function identification

- Prologues and Epilogs
- Function Padding
- Data references
- Exception handling references
- Direct branch exploration




Table lookup considerations

- Tens of thousands of indirect branch mispredictions happen every minute
- Our CFI policy look up must be very fast and we take into consideration search is more important than insertion
- We decided on AVL trees which provide $O(\log n)$ search to maximize performance
- Any additional processing is sent down a “slow-path” which operates outside of the ISR



CFI takeaways

- A violation of CFI occurs when a destination address is not found in our table of discovered functions
- Every image loaded into a protected process is dynamically processed to build the CFI table
- When an indirect branch is mispredicted our kernel driver can compare the destination against our list
- Since the system is interrupted we have a great opportunity to prevent the attack from taking place if the destination is not allowed

Efficacy

CODE EXECUTION TECHNIQUE	SAMPLES	HA-CFI DETECT RATE	EMET DETECT RATE
ROP	37	95%	100%
ROPless Technique A	1	100%	0%
ROPless Technique B	10	100%	0%

Efficacy

BUG CLASS	# CVE'S	# SAMPL ES	HA-CFI DETECTION RATE
Out-of-bounds Write	3	6	83.3%
Buffer Overflow	3	6	83.3%
Integer Overflow	2	6	100%
Use-After-Free	4	14	100%
Double Free	2	4	100%
Type Confusion	3	6	100%
Race Condition	1	4	100%
Uninitialized Memory	1	1	100%



Conclusion

- Exploit techniques change rapidly
- Attackers use creativity to bypass “post-exploit” protections
- Exploit defense needs to detect and prevent exploitation at the earliest phase
- Compile-time solutions are powerful but not sufficient
- Hardware can be utilized to enforce CFI policies at runtime
- Real prevention requires defense-in-depth

Questions?

Jamie: @jamierbutler

Cody: @CodyPierce