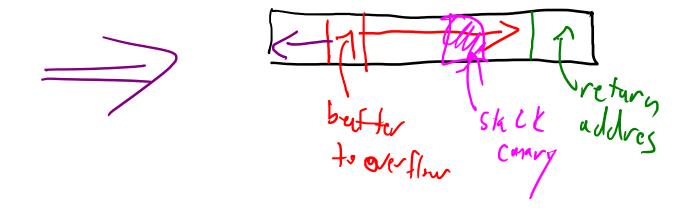
EXERCISE #17

MEMORY ATTACK REVIEW

Write your name and answer the following on a piece of paper

Describe how a stack canary protects against return-oriented programming



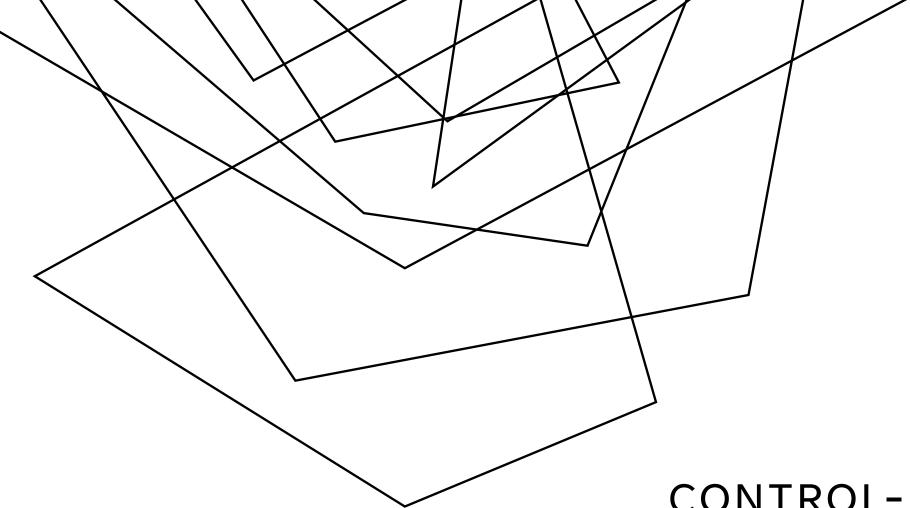
Second reading assigned

- The original paper on CFI

Basically halfway through the semester

- Time to check in on how things are going

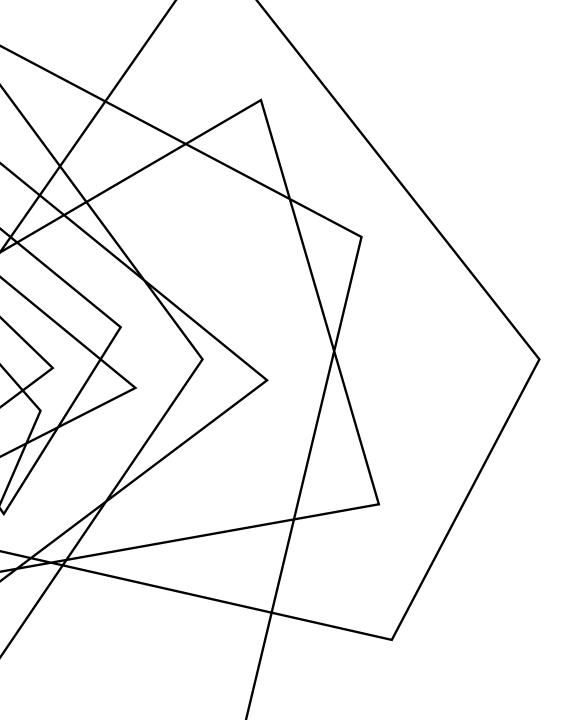
ADMINISTRIVIA AND ANNOUNCEMENTS



CONTROL-FLOW INTEGRITY

EECS 677: Software Security Evaluation

Drew Davidson



TOPIC CONTEXT

CONTEMPLATED A FORM OF ATTACK, LEFT WITH A HINT OF DEFENSES

LAST TIME: MEMORY ATTACKS

REVIEW: LAST LECTURE

BUFFER OVERFLOWS

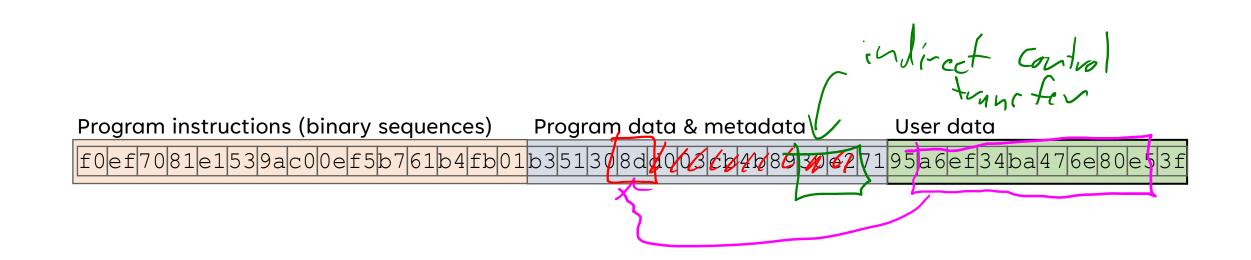
Exceed the boundary of a region of memory, start overwriting other program (meta)data

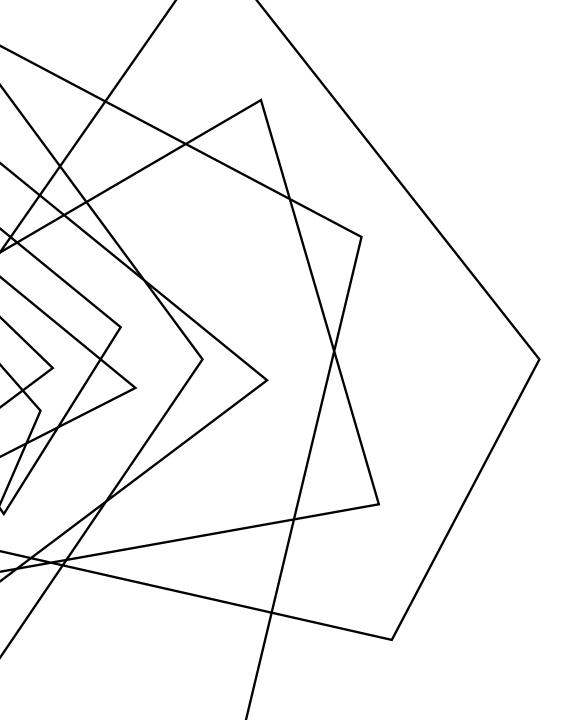
CODE INJECTION

Overwrite a return address and jump to your own user-written buffer

RETURN-ORIENTED PROGRAMMING

Overwrite a return address and jump to "gadgets" of existing code





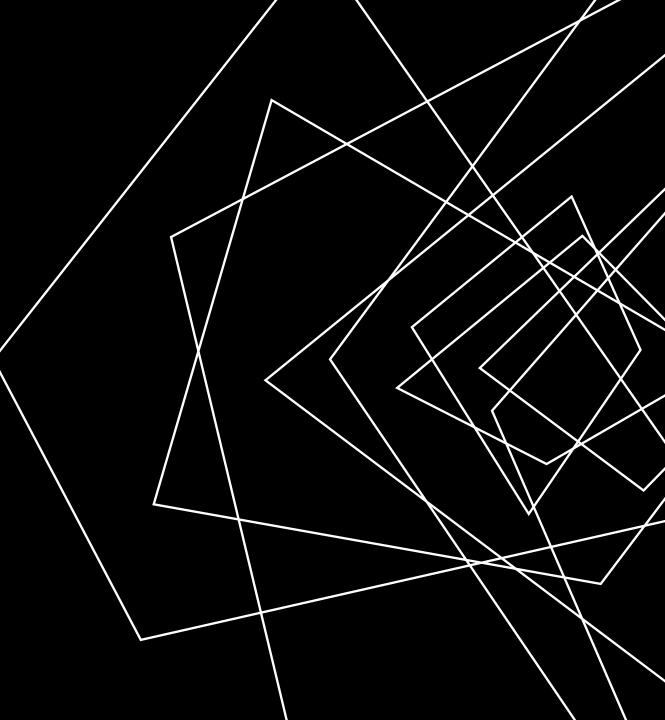
OVERVIEW

KEEP THE CONTROL FLOW "ON RAILS"



LECTURE OUTLINE

- Motivation
- Implementation considerations
- Practical manifestations



WE KNOW THE PROBLEM

JUMPING WHERE YOU SHOULDN'T

- This certainly includes ROP
- Might also involve other attacks

```
#include <string.h>
struct auth {
       char pass[4];
        void (*func)(struct auth*);
};
void success() { printf("Success!\n"); }
void failure() { printf("Failure\n"); }
void check(struct auth *a) {
        if (strcmp(a->pass, "pass") == 0)
                a->func = &success;
       else
                a->func = &failure;
int main(int argc, char **argv) {
        struct auth a;
        printf("Enter your password:\n");
        scanf("%s", &a.pass);
        a.func(&a);
```

#include <stdio.h>

WE KNOW THE PROBLEM

JUMPING WHERE YOU SHOULDN'T

- This certainly includes ROP
- Might also involve other attacks

LOOK, NO RET OVERWRITE!

```
#include <stdio.h>
#include <string.h>
struct auth {
       char pass[4];
        void (*func)(struct auth*);
};
void success() { printf("Success!\n"); }
void failure() { printf("Failure\n"); }
void check(struct auth *a) {
        if (strcmp(a->pass, "pass") == 0)
                a->func = &success;
        else
                a->func = &failure;
int main(int argc, char **argv) {
        struct auth a;
        printf("Enter your password:\n");
        scanf("%s", &a.pass);
        a.func(&a);
```

WE KNOW THE PROBLEM

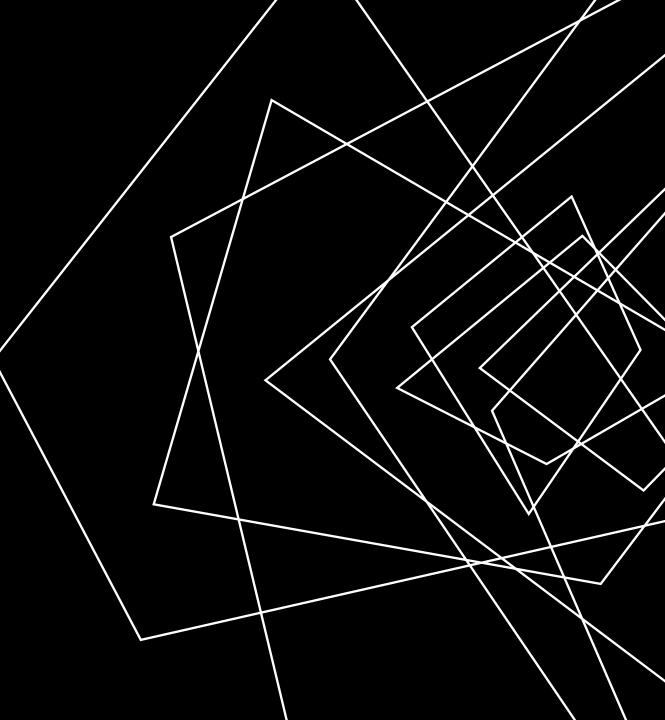
JUMPING WHERE YOU SHOULDN'T

- This certainly includes ROP
- Might also involve other attacks

LOOK, NO RET OVERWRITE!

LECTURE OUTLINE

- Motivation
- Implementation considerations
- Practical manifestations



foo -> bar -> baz

NAÏVE APPROACH:

Encode the entire CFG into the program text

CALL GRAPH ANALYSIS IMPLEMENTATION CONSIDERATIONS

NAÏVE APPROACH:

Encode the entire CFG into the program text

NAÏVE APPROACH:

Encode the entire CFG into the program text

SSUES:

Dynamic: overhead

Static: precision

NAÏVE APPROACH:

Encode the entire CFG into the program text

ISSUES:

Dynamic: overhead

NAÏVE APPROACH:

Encode the entire CFG into the program text

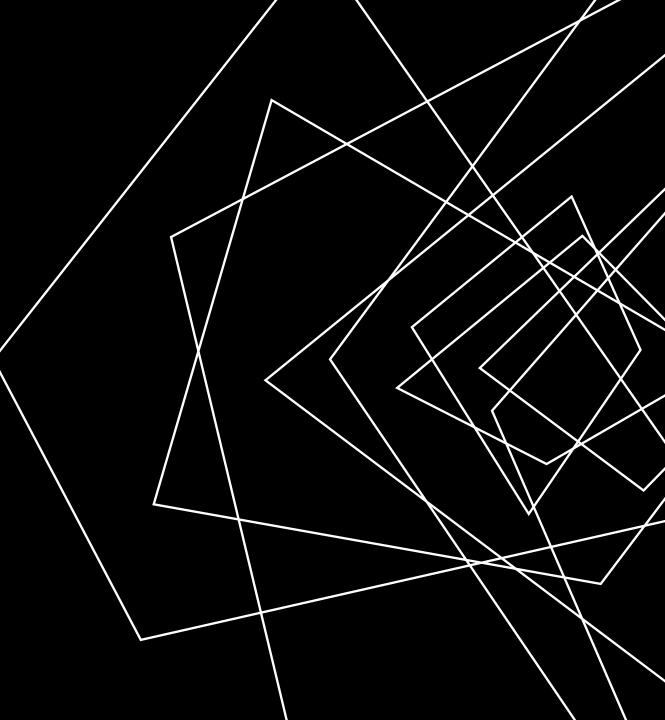
SSUES:

Dynamic: overhead

Static: precision

LECTURE OUTLINE

- Motivation
- Implementation considerations
- Practical manifestations



INTEL CET PRACTICAL MANIFESTATIONS

CONTROL-FLOW ENHANCEMENT TECHNOLOGY

Requires recompilation of software to support

Requires hardware support (!)

Scope

1) Prevent ret overwriting with a shadow stack

low addresser regular process ref Shylow dalk

INTEL CET PRACTICAL MANIFESTATIONS

CONTROL-FLOW ENHANCEMENT TECHNOLOGY

Requires recompilation of software to support

Requires hardware support (!)

Scope

1) Prevent ret overwriting with a shadow stack

2) Hardware modifications prevent indirect jungs its redjels

INTEL CET PRACTICAL MANIFESTATIONS

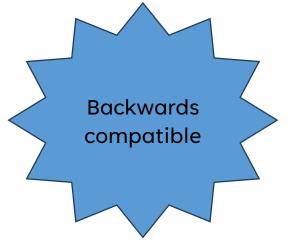
CET HARDWARE CHANGES

Altered semantics of the CALL and JMP

Moves a processor state machine into the WAIT_FOR_ENDBRANCH state In WAIT_FOR_ENDBRANCH, next instruction must be the ENDBRANCH instruction

Added a new instruction at control-transfer targets





MICROSOFT CONTROL FLOW GUARD

figuration: Active(Debug)	 Platform: Active(Win32) 	~	Configuration Manager
 ▲ Configuration Properties ▲ General Debugging VC++ Directories ▲ C/C++ General Optimization Preprocessor Code Generation Language Precompiled Heade Output Files Browse Information Advanced 	Enable String Pooling		
	Enable Minimal Rebuild	Yes (/Gm)	
	Enable C++ Exceptions	Yes (/EHsc)	
	Smaller Type Check	No	
	Basic Runtime Checks	Both (/RTC1, equiv. to /RTCsu) (/RTC1))
	Runtime Library	Multi-threaded Debug DLL (/MDd)	,
	Struct Member Alignment	Default	
	Security Check	Enable Security Check (/GS)	
	Control Flow Guard	Yes (/guard:cf)	~
	Enable Function-Level Linking		
	Enable Parallel Code Generation		
	Enable Enhanced Instruction Set	Not Set	
	Floating Point Model	Precise (/fp:precise)	
	Enable Floating Point Exceptions		
All Options Command Line	Create Hotpatchable Image		
Linker	ereate notpatenable image		
General			
Input			
Manifest File			
Debugging			
System			
Optimization			
Each and deal UDI	Control Flow Guard Guard security check helps detect attemp	ts to dispatch to illegal block of code. (/guard	d:cf)





RECALL FROM LAST TIME...

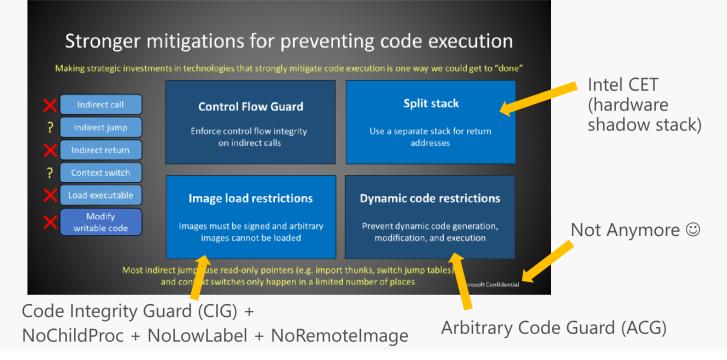
ROP attacks considered harmful

HOW INDUSTRY RESPONDED

MS CFG as a case study in a lot of interesting aspects of software security



2012 Strategy Slide Deck



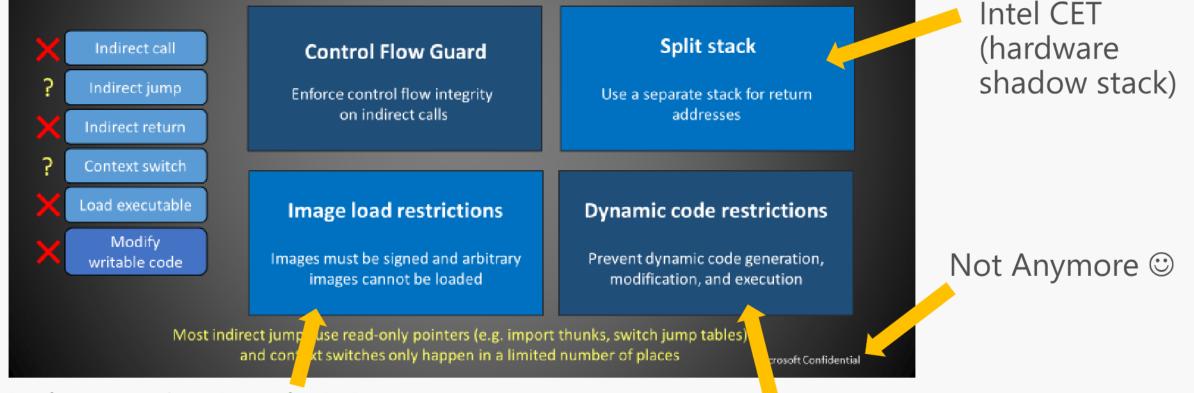
Source: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/ 2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf

2012 Strategy Slide Deck



Stronger mitigations for preventing code execution

Making strategic investments in technologies that strongly mitigate code execution is one way we could get to "done"



Code Integrity Guard (CIG) + NoChildProc + NoLowLabel + NoRemoteImage

Arbitrary Code Guard (ACG)



THIS IS AN INTERESTING TALK!

I'd recommend you watch it: <u>https://www.youtube.com/watch?v=oOqpI-2rMTw</u>

IT COMES WITH THE HISTORICAL BURDEN OF CONTROL FLOW GUARD

Widely-publicized issue that allowed it to be avoided

Theory



Microsoft's overarching goal is to make exploitation financially infeasible or impossible

All RCE memory corruption exploits found in-the-wild hijack control flow



Attackers often follow "path of least resistance", breaking them means increasing cost of exploitation Constraining control flow to "legitimate" paths breaks all of these exploits aswritten

After some formal thought, we believe CFI will robustly mitigate against stronger primitives



Security teams are well positioned to drive these changes

CFG had no formal threat model during very early development. Thought of as a way to kill ROP.

Hindsight is 20/20, but we did have formal thought around future exploit trends. See [1]



CONTROL FLOW GUARD HAS A HISTORICAL BURDEN

Widely-publicized issue that allowed it to be avoided

We'll get to the actual workaround, but let's talk about its impact



CONTROL FLOW GUARD PRACTICAL MANIFESTATIONS

DETAILS

Precision: call needs to be a valid function entry point

Enforcement: OS verifies indirect control transfer destinations via a table in protected memory

PROTECTIONS

Protected destinations page in read-only memory Read-only memory bit can be turned off by attacker ③

CLANG'S CFI PRACTICAL MANIFESTATIONS

DETAILS

Precision: call needs to match type signature

Enforcement: compiler-inserted checks

WRAP-UP

