Software Defense

Mustakimur R. Khandaker

Control Flow Hijacking

Attacker's goal: take over target machine (e.g., web server).
- Execute arbitrary code on target by hijacking control flow.

Common vulnerabilities:
- Buffer overflow.
- Integer overflow.
- Format string vulnerabilities.
- UAF and Double-free.

Common targets:
- Return address.
- Vtable.
- Function pointer.

Common exploitations:
- Code injection (shellcode).
- Return-to-libc.
- Return-oriented Programming (ROP).
- Heap spraying.

What Makes a Process Safe?

Control-flow safety: all control transfers are **“possible”** by the original program.

- No arbitrary jumps, no calls to library routines that the original program did not call.

Memory safety: all memory accesses are **“correct”**.

- Respect array bounds, don’t stomp on another process’s memory, separation between code and data.

Type safety: all function calls and operations have arguments of **correct** type.

Defend Control Flow Hijacking

Fix bugs:

- Audit software (Review).
  - Automated tools: Coverity, Prefast/Prefix.
- Identify vulnerabilities.
  - Marketplace, fuzzing tools.
- Rewrite software in a type safe language (Java, Rust).
  - Difficult for existing (legacy) code.

Concede overflow, but prevent code execution (exploitation).

Add runtime code to detect exploits:

- Halt process when overflow exploit detected.
  - StackGuard, LibSafe, ...

Canary/Stack Cookies

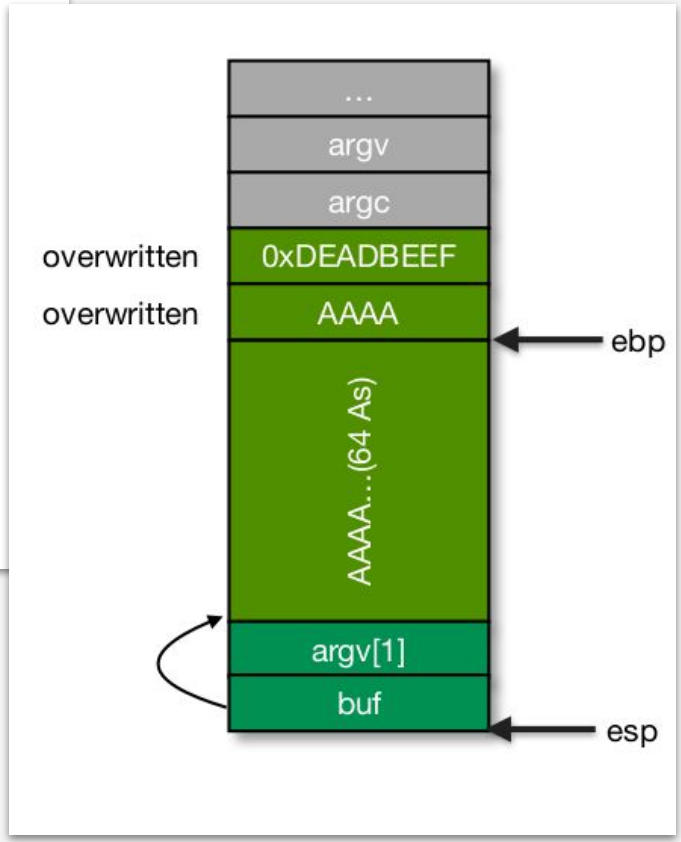
Stack Canary

"**Canary in a coal mine**" is frequently used to refer to a person or thing which serves as an early warning of a coming crisis.

- **wikipedia**

Stack-based Buffer Overflow

```c
#include <string.h>
int main(int argc, char **argv)
{
    char buf[64];
    strcpy(buf, argv[1]);
    return 0;
}
```

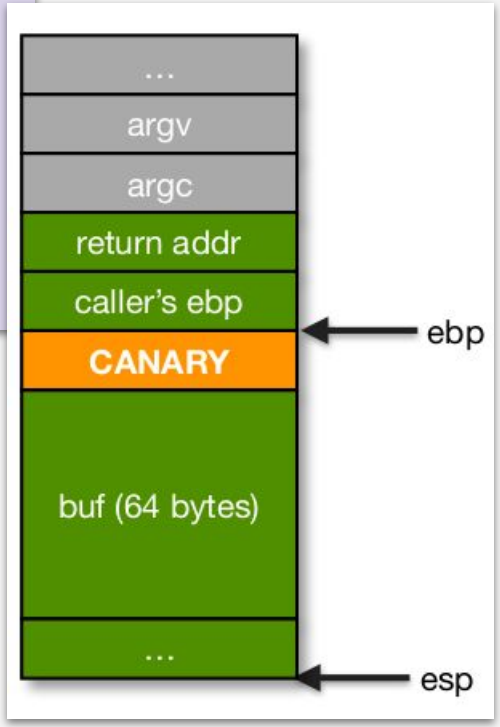


StackGuard (Cowen etal. 1998)

Idea:

- Function prologue embeds a canary word between return address and locals.
- Function epilogue checks canary before it returns.

wrong canary ➔ overflow



Continue …

StackGuard implemented as a **GCC** patch.

- Program must be recompiled.
- Minimal performance effects: 8% for Apache.
- **Note:** Canaries do not provide full protection.
  - Some stack smashing attacks leave canaries unchanged.
  - By leveraging some form of information leak.

Heap protection: *PointGuard*.

- Protects function pointers and setjmp buffers by encrypting them.
  - e.g. XOR with random cookie.
- Less effective, more noticeable performance effects.

Canary Types

Random canary:

- Random string chosen at program startup.
- Insert canary string into every stack frame.
- Verify canary before returning from function.
  - Exit program if canary changed. Turns potential exploit into DoS.
- To corrupt, attacker must learn current random string.

Terminator canary:

Canary = {0, newline, linefeed, EOF}

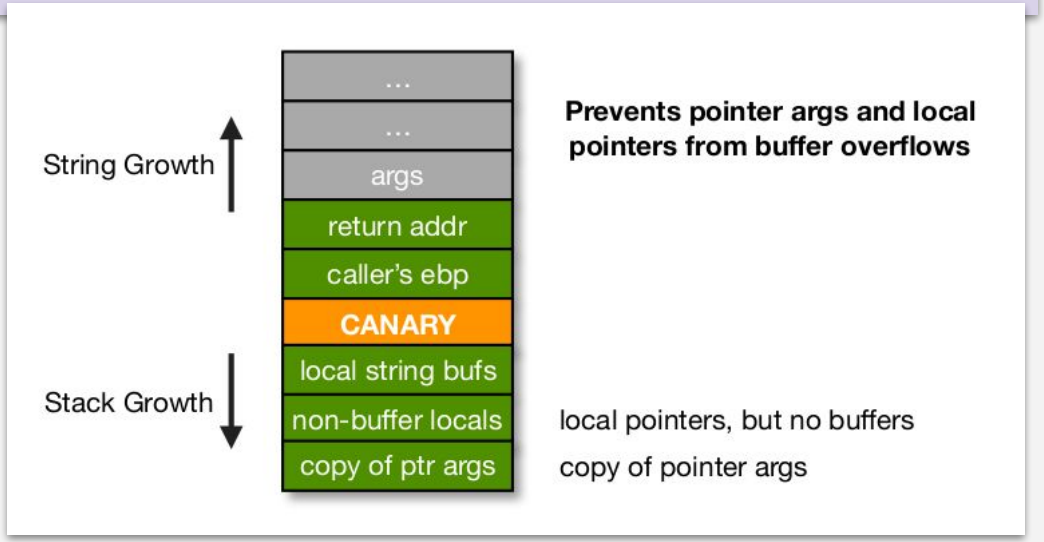
- String functions will not copy beyond terminator.
- Attacker cannot use string functions to corrupt stack.
  - How about ***memcpy()***.

StackGuard Improvement: ProPolice

ProPolice (IBM) - gcc 3.4.1. *(-fstack-protector).*

It improved on the idea of StackGuard by placing buffers after local pointers and function arguments in the stack frame.

This helped avoid the corruption of pointers, preventing access to arbitrary memory locations.



GCC Stack-Smashing Protector

```
…
;Function Prologue, Embed Canary
mov     %gs:20,%eax
mov     %eax,-4(%ebp)
xor     %eax,%eax

…
```

```
;Function Epilogue, Verify Canary
mov     -4(%ebp),%edx
xor     %gs:20,%edx
je      0x8048477 <main+55>
call    0x8048340 <__stack_chk_fail@plt>
leave
ret
```

Compiled with gcc v4.6.1:
gcc **-fstack-protector** -O1 …

%gs:20 ???
xor %eax, %eax ???

MS Visual Studio /GS (since 2003)

Compiler **/GS** option:

- Combination of ProPolice and random canary.
- If cookie mismatch, default behavior is to call _exit(3).

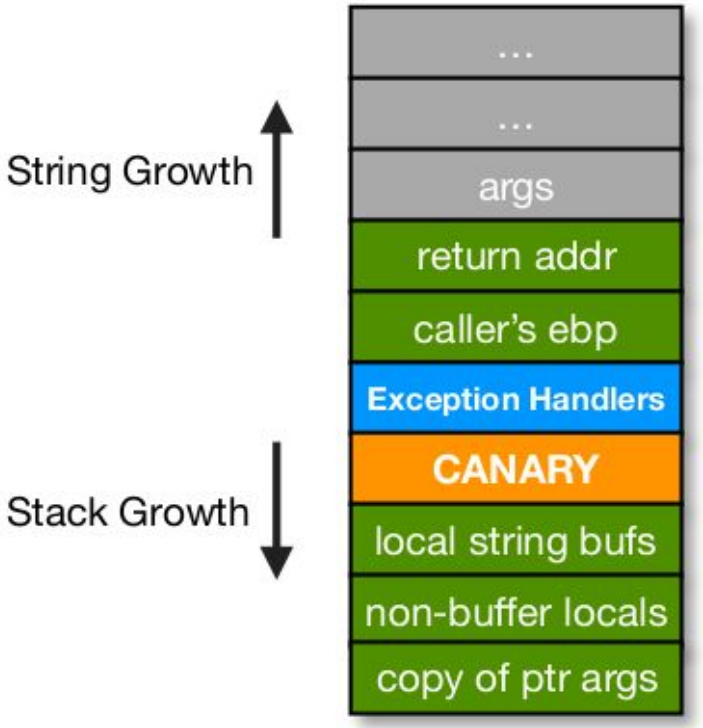
Enhanced **/GS** in Visual Studio 2010:

- /GS protection added to all functions, unless can be proven unnecessary.

```
;Function prologue, embed canary
sub    esp, 8
mov    eax, DWORD PTR ___security_cookie
xor    eax, esp
mov    DWORD PTR [esp+8], eax
```

```
;Function epilogue, verify canary
mov    ecx, DWORD PTR  [esp+8]
xor    ecx, esp
call   @__security_check_cookie@4
add    esp, 8
```

/GS Stack Frame



Canary protects return addr and exception handler frame

local pointers, but no buffers

copy of pointer args

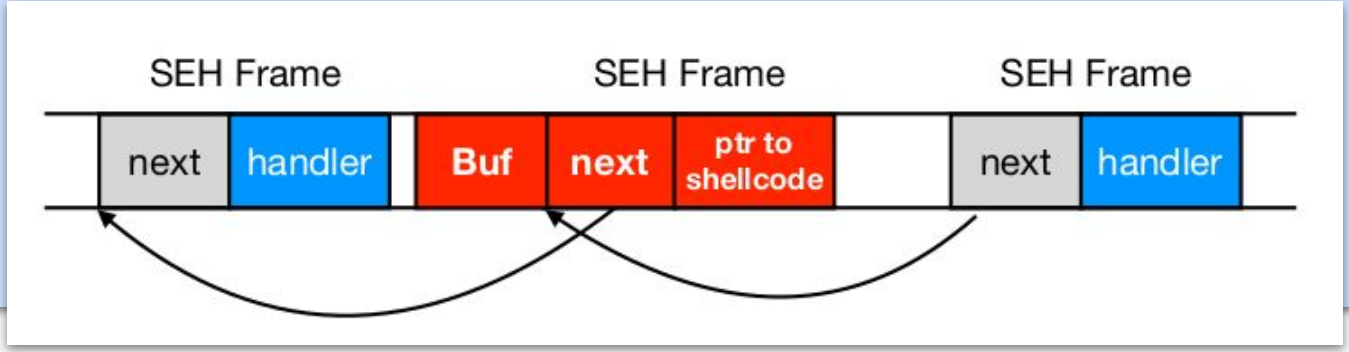
Evading /GS with Exception Handlers

To evade **/GS**,

- Overflow exception handler to the shellcode.
- Trigger the exception to hijack the control flow.

When exception is thrown, dispatcher walks up exception list until a handler is found (else use default handler).

Main point: exception is triggered before canary is checked.



Defenses: SAFESEH and SEHOP

SAFESEH: linker flag.

- Linker produces a binary with a table of safe exception handlers.
- System will not jump to exception handler not on list.

/SEHOP: platform defense (since win vista SP1).

- **Observation:** SEH attacks typically corrupt the “next” entry in SEH list.
- **SEHOP:** add a dummy record at top of SEH list
- When exception occurs, dispatcher walks up list and verifies dummy record is there. If not, terminates process.

Summary: Canaries are not Full Proof

Canaries are an important defense tool, but do not prevent all control hijacking attacks:

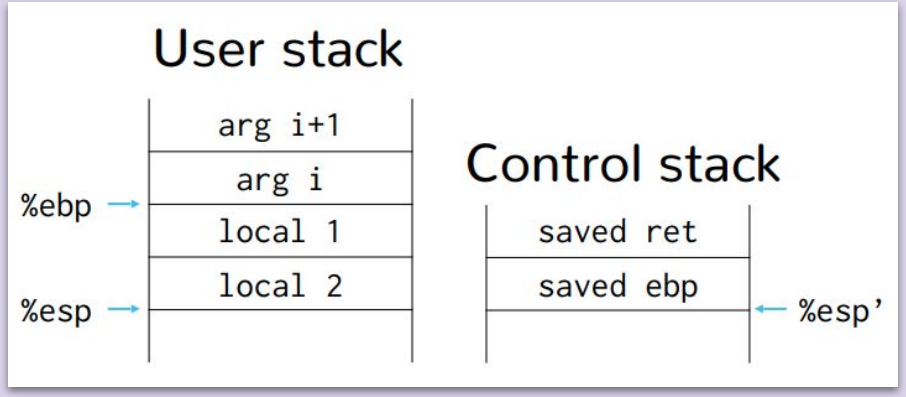
- Heap-based overflows are still possible.
- Integer overflows are still possible.
- Use-after-free can mount arbitrary code execution.
- /GS by itself does not prevent Exception Handling attacks (also need SAFESEH and SEHOP).

Shadow Stack/
Safe Stack

Separate Control Stack

Problem: The stack smashing attacks take advantage of the weird machine: control data is stored next to user data.

Solution: Make it less weird by bridging the implementation and abstraction gap: separate the control stack.

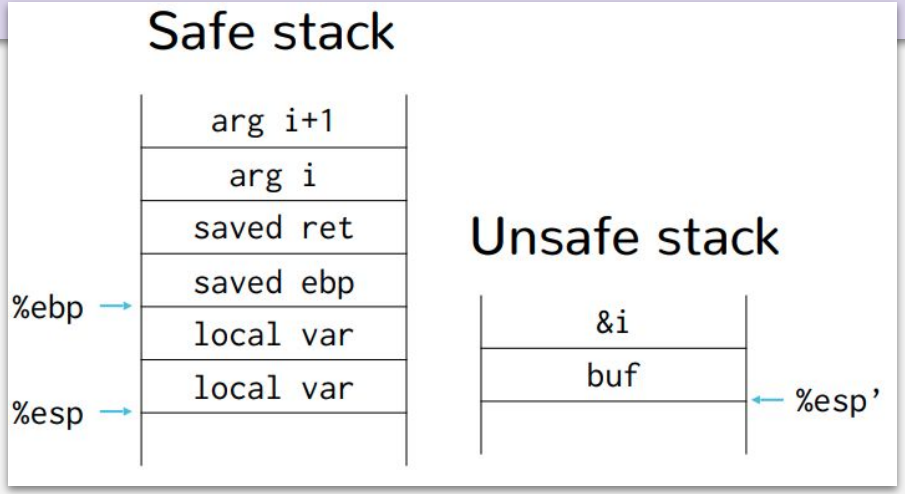


WebAssembly (Wasm) has a separate stack.

Safe Stack

"SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack.

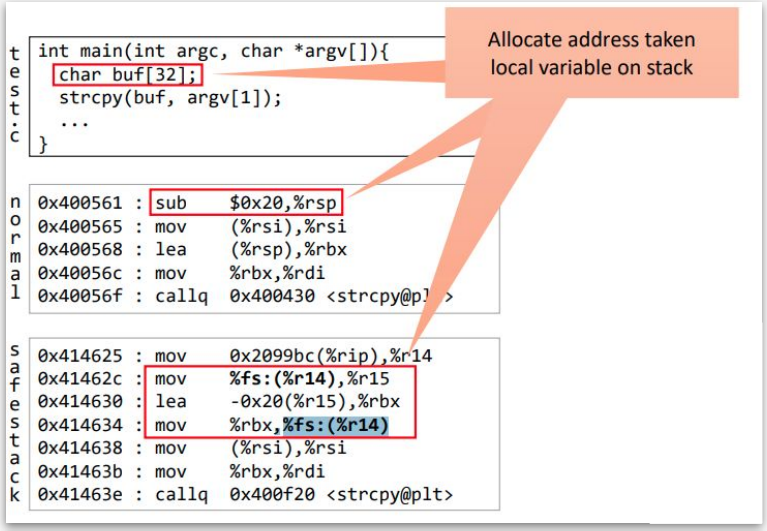
The safe stack stores return addresses, register spills, and local variables that are always accessed in a safe way, while the unsafe stack stores everything else. This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack."



Implementation

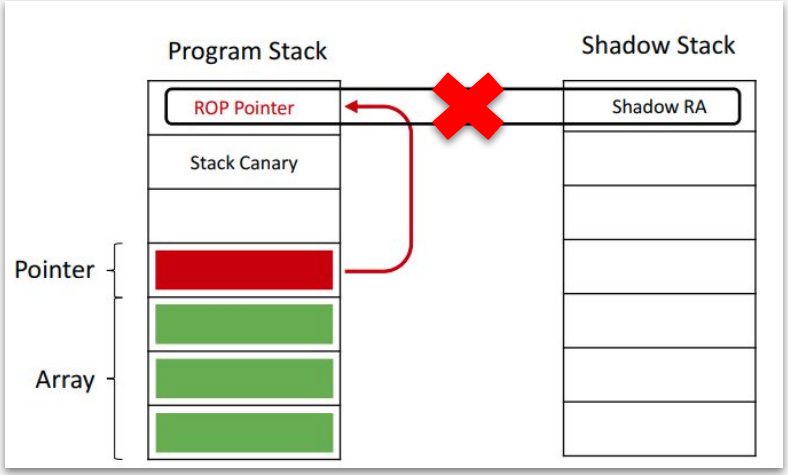
There is no actual separate stack, we only have linear memory and loads/store instructions.

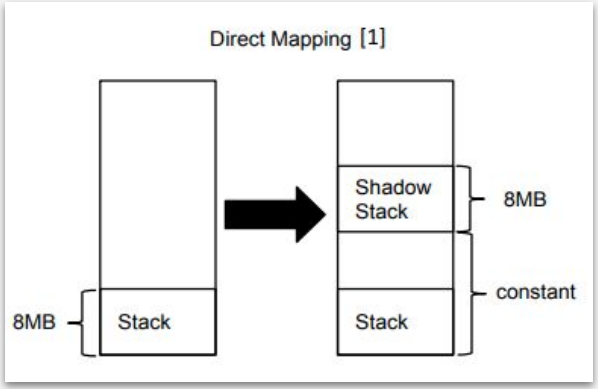
- Compile time instrumentation pass.
  - **Flag:** -fsanitize=safe-stack.
- Put the safe stack in a random place in the address space.
  - **Assumption:** location of control/stack stack is secret.
  - Relies on ASLR.
- Ensure stack access is “safe”.
  - Address taken objects moved to alternative stack.

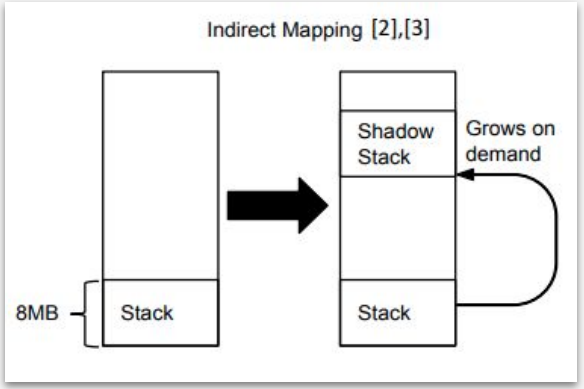


*Intel calling convention RDI, RSI.

Shadow Stack





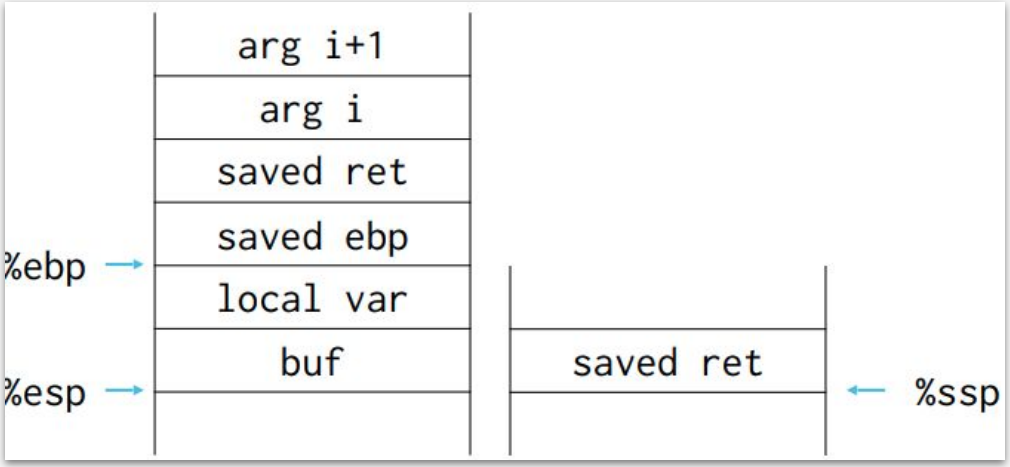


Intel's Upcoming Shadow Stack

Addresses both the performance and security issues.

- New shadow stack pointer (%ssp).
- call and ret automatically update %esp and %ssp.
- Can't update shadow stack manually.

Note: May need to rewrite code that manipulates stack manually.



W^X / DEP

Write XOR Execute (W ^ X)

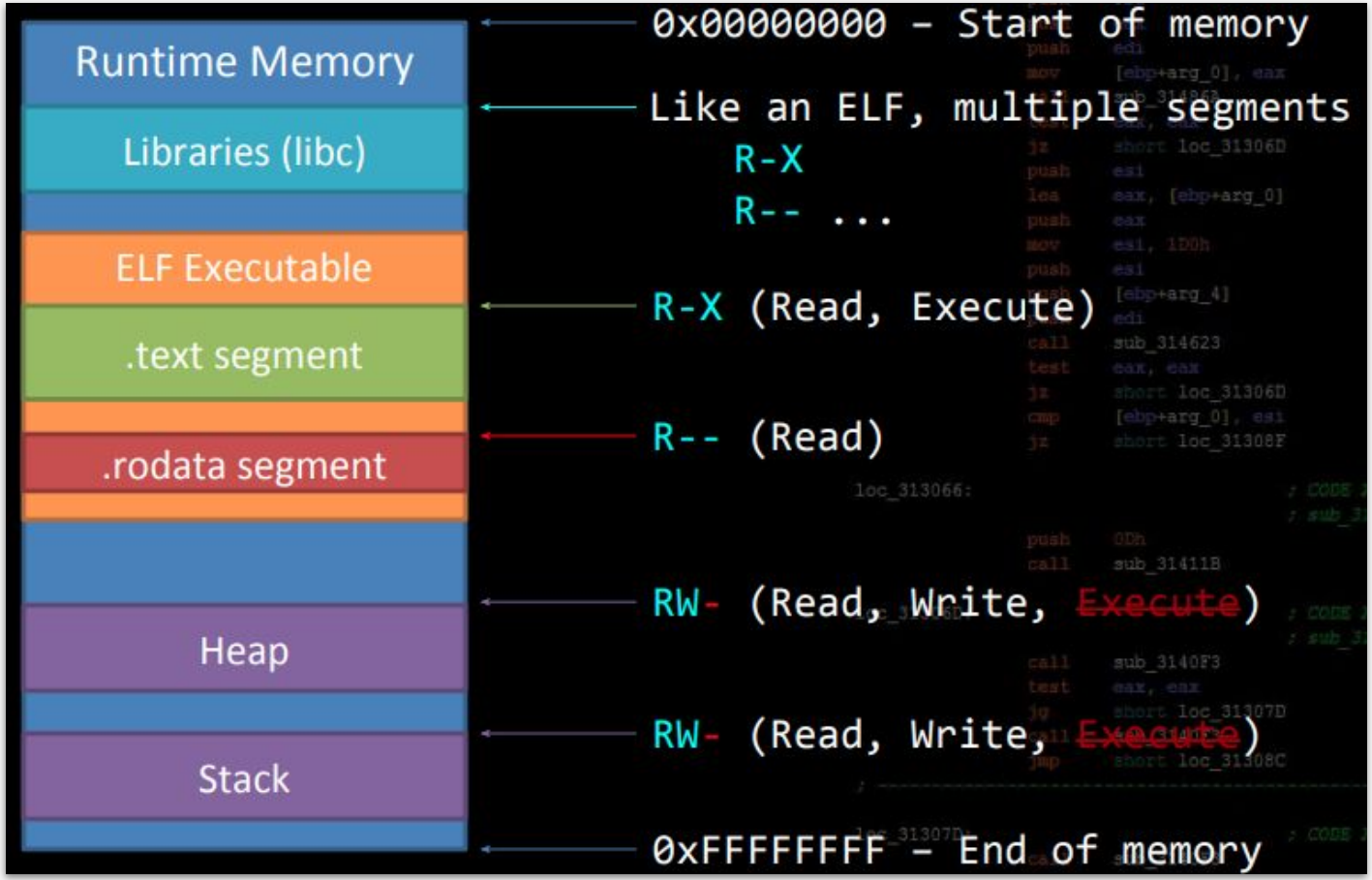
Use hardware memory protection to ensure memory cannot be both writeable and executable at same time.

- Code is executable, not writeable.
- Stack, heap, static variables writeable, not executable.
- Supported by most modern processors.
  - CPU supports NX bit in every page table entry.
- Implemented by modern operating systems.

Limitations:

- Some apps need executable heap (e.g., JITs).
  - e.g., heap spraying attacks.
- Does not defend against code reuse attacks.
  - return-to-libc and return-oriented programming attacks.

Layout


Runtime Memory
Libraries (libc)
ELF Executable
.text segment
.rodata segment
Heap
Stack
0x00000000 – Start of memory
Like an ELF, multiple segments
R-X
R-- ...
R-X (Read, Execute)
R-- (Read)
RW- (Read, Write, Execute)
RW- (Read, Write, Execute)
0xFFFFFFFF – End of memory

Examples: DEP Controls in Windows

Address Randomization

Address Space Randomization

Traditional exploits need precise addresses.
- Stack-based buffer overflows: location of the shellcode.
- Return-to-libc: library function addresses.

Problem: a program's memory layout is fixed/predictable.
- Code, stack, heap, BSS, ...

Solution: randomize the memory layout of a running process.
- Position-independent code that can be loaded at any locations.
- Random allocation for the stack, heap, and BSS.
- Called **Address Space Layout Randomization (ASLR)**.

ASLR Example

Running cat twice.

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000 rw-p 082ac000 00:00 0          [heap]
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf966000-bf97b000 rw-p bffeb000 00:00 0          [stack]
```

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
086e8000-08709000 rw-p 086e8000 00:00 0          [heap]
b7d9a000-b7eef000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7eef000-b7ef0000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7ef0000-b7ef2000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf902000-bf917000 rw-p bffeb000 00:00 0          [stack]
```

Address Space Layout

On 32-bit architectures, ASLR has limited entropy.

- 16 bit of randomness for base address a and b.
- 24 bit of randomness for base address c.

Ubuntu - ASLR

ASLR is **ON** by default [Ubuntu-Security]

- *cat /proc/sys/kernel/randomize_va_space*
  Prior to Ubuntu 8.10: 1 (stack/mmap ASLR)
  In later releases: 2 (stack/mmap/brk ASLR)

  Requires Position Independent Executable (PIE)
  PIE is compiled with *"gcc -fPIE –pie"*

```
0000000000400470 <.plt>:
  400470:   ff 35 7a 1b 00 00       push   QWORD PTR [rip+0x1b7a]        # 401ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
  400476:   f2 ff 25 7b 1b 00 00    bnd jmp QWORD PTR [rip+0x1b7b]        # 401ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
  40047d:   0f 1f 00                nop    DWORD PTR [rax]
  400480:   f3 0f 1e fa             endbr64
  400484:   68 00 00 00 00          push   0x0
  400489:   f2 e9 e1 ff ff ff       bnd jmp 400470 <_init+0x20>
  40048f:   90                      nop
  400490:   f3 0f 1e fa             endbr64
  400494:   f2 ff 25 65 1b 00 00    bnd jmp QWORD PTR [rip+0x1b65]        # 402000 <puts@GLIBC_2.2.5>
  40049b:   0f 1f 04 00             nop    DWORD PTR [rax+rax*1]
  40049f:   90                      nop
```

Compiled with **-no-pie**

```
Breakpoint 1 at 0x401126: file main.c, line 4.

Breakpoint 1, main () at main.c:4
4           puts("hello");
pc = 0x401126

Breakpoint 1, main () at main.c:4
4           puts("hello");
pc = 0x401126
```

Compiled with **-pie**

```
Breakpoint 1 at 0x1139: file main.c, line 4.

Breakpoint 1, main () at main.c:4
4           puts("hello");
pc = 0x5630df2d6139

Breakpoint 1, main () at main.c:4
4           puts("hello");
pc = 0x55763ab2e139
```

Bypassing ASLR

Brute-force attacks:
- Try many times, eventually get lucky.
- e.g., Blind ROP (BROP).

Use ROP to exploit **non-randomized memory (code/data).**
- Code (program or libraries) that is NOT compiled as **PIE**.
- Systems that have ASLR off by default for “compatibility”.

Exploit **information disclosure** bugs to reveal addresses.
- ASLR only randomizes code/data segment bases.
- e.g., **JIT-ROP.**

Code Randomization beyond ASLR

ASLR randomizes memory layout.

- Base addresses of code, data, and shared libraries.
- ASLR is the only widely deployed.

Other randomization techniques:

- **System call randomization:** randomize syscall id's.
- **Instruction set randomization.**
- **Fine-grained code randomization:**

  function-level, basic block level, instruction-level (in-place code randomization).

- **Live randomization** (re-randomization).

SoK: Eternal War in Memory
IEEE Symposium on Security and Privacy (IEEE S&P) May, 2013.

SoK: Automated Software Diversity
IEEE Symposium on Security and Privacy (IEEE S&P) May, 2014

Obfuscation

Code Obfuscation

Code obfuscation is the generation or alteration of source code and/or object code in such a way that it is easy for the computer to comprehend but considerably difficult to reverse engineer.

Main features:

- **Potency:** is the level up to which a human reader would be confused by the new code.
- **Resilience:** is how well the obfuscated code resists attacks by deobfuscation tools.
- **Cost:** is how much load is added to the application.

Methods

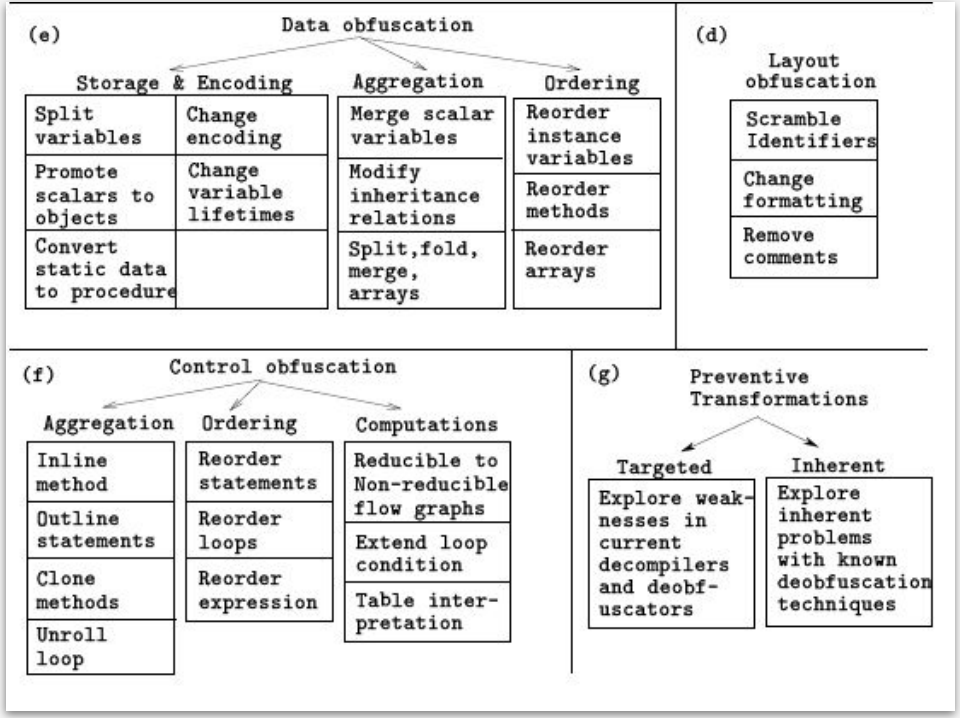
Lexical transformations.
- Modify variable names.

Control transformations.
- Change program flow while preserving semantics.

Data transformations.
- Modify data structures.

Anti-disassembly.

Anti-debugging.



Example

Original code:

```
function foo( arg1)
{
  var myVar1 =  "some string";  //first
comment
  var intVar =  24 *  3600; //second comment
  /* here is
a long
multi-line comment blah */
  document. write( "vars are:" +  myVar1 +  "
" +  intVar +  " " +  arg1) ;
} ;
```

Obfuscated code:

```
function z001c775808( z3833986e2c) { var
z0d8bd8ba25=
"\x73\x6f\x6d\x65\x20\x73\x74\x72\x69\x6e\
x67"; var z0ed9bcbcc2= (0x90b+785-0xc04)*
(0x1136+6437-0x1c4b); document. write(
"\x76\x61\x72\x73\x20\x61\x72\x65\x3a"+
z0d8bd8ba25+ "\x20"+ z0ed9bcbcc2+ "\x20"+
z3833986e2c);};
```

Reference Monitor

Reference Monitor

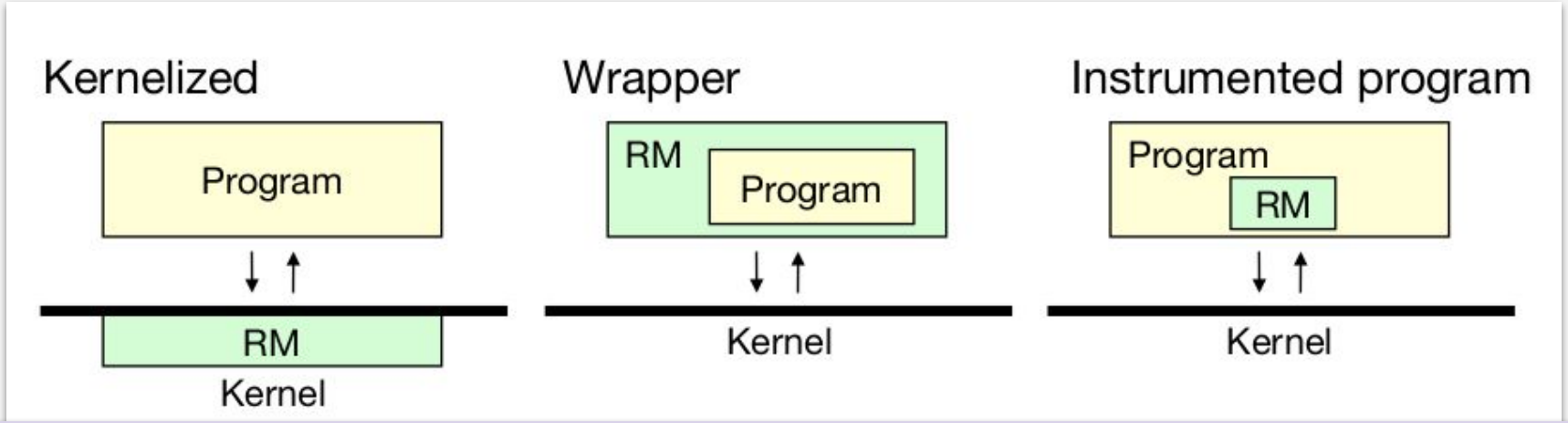
Observe execution of the program/process.

- At what level? possibilities: **hardware, OS, network.**

Halt or contain execution if the process violates the policy.

- e.g., program safety policy.
- Which system events are relevant to the policy?
  - Instructions, memory accesses, system calls, network packets ...

Reference Monitor Placement



Example:

- In-kernel RM: **syscall interposition**.
- Wrapper: **program shepherding**.
- In-program RM (inline reference monitor): **CFI, SFI, Native Client**.

OS as a Reference Monitor

Collection of running processes and files.

- Processes are associated with users.
- Files have **Access Control Lists (ACLs)** saying which users can read/write/execute them.

OS enforces a variety of safety policies.

- File accesses are checked against file's ACL.
- Process cannot write into memory of another process (isolation).
- Some operations require superuser privileges (privilege).
- Enforce CPU sharing, disk quotas, etc. (sharing).

Inline Reference Monitor

IRM: instrument program code to enforce expected behavior.

- Example: control flow integrity, software fault isolation ...

Principles:

- **Complete mediation:** reference monitors must always be invoked.
- **Tamper-proof:** reference monitors cannot be modified by attackers.
- **Verifiable correctness:** references monitor's correctness, preferably, verifiable.
- Performance and compatibility.

Control Flow Integrity (CFI)

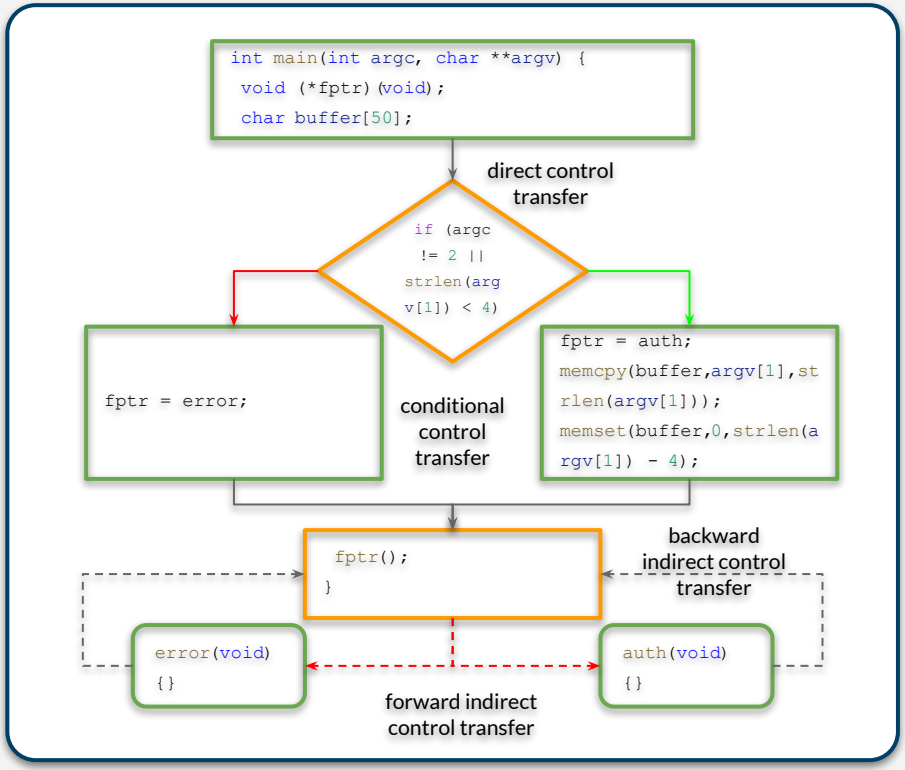
Control Flow Hijack

```
void error(void) {}
void auth(void) {}
int main(int argc, char **argv) {
  void (*fptr)(void);
  char buffer[50];

  if (argc != 2 || strlen(argv[1]) < 4) {
    fptr = error;
  } else {
    fptr = auth;
    memcpy(buffer,argv[1],strlen(argv[1]));
    memset(buffer,0,strlen(argv[1]) - 4);
  }

  fptr();
}
```

Control-flow hijack:
i) Privilege escalation.
ii) Information leak.



Control-flow Integrity (CFI)

CFI is a defense mechanism against control-flow hijacking that employs inline reference monitor to enforce the run-time control flow of a process must follow the statically computed control-flow graph (CFG).

Steps:

- Compute static CFG of the program (compile time).
- Instrument program code with IRM (compile time).
  - IRM encodes the program's CFG in some way.
- Inserted monitors ensure runtime execution always stays within the statically determined CFG (runtime).

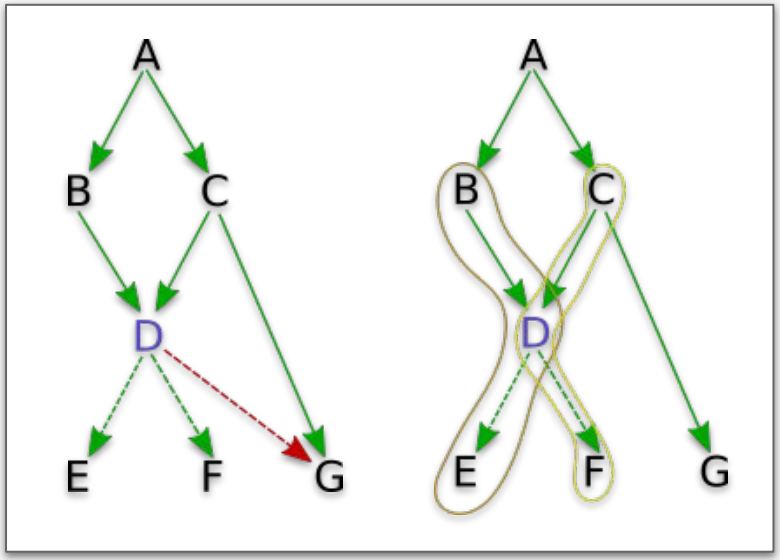
Goal: prevent injection of arbitrary code and invalid control transfers (e.g., return-to-libc).

- Secure even if the attacker has complete control over the thread's address space.

CFI Policy

CFI policies are categorized into two:

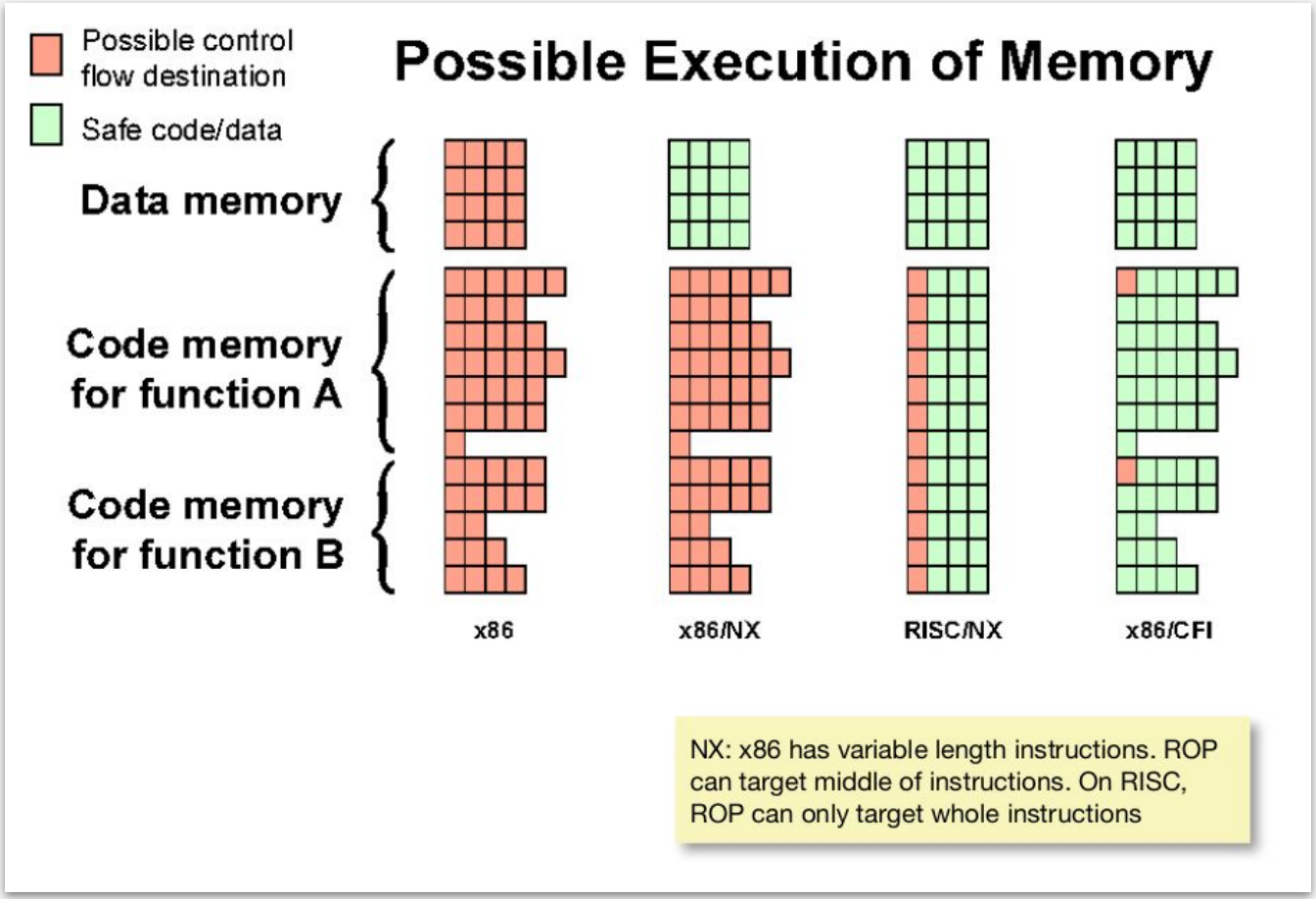
- **Context-insensitive (CI-) CFI:** CFI policy without additional information.
  - e.g. type-based, address-taken etc.
- **Context-sensitive (CS-) CFI:** CFI policy with past execution history.
  - e.g., path sensitivity.



Effectiveness of CFI can be measured by the largest and average EC size

➢ An EC is a group of targets that CFI cannot distinguish/separate

Possible Execution of Memory

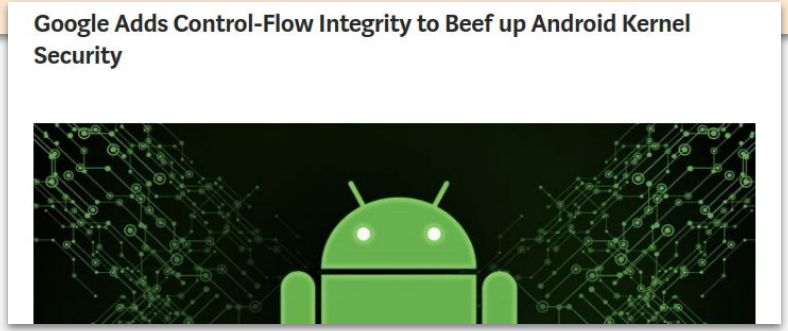


Real World Adoption (CFI)

Intel Control-flow Enforcement Technology:
Native support to use a shadow stack and **Indirect Branch Tracking (IBT).**

Clang CFI:
Number of available schemes are available on Clang e.g. cfi-cast-strict, cfi-vcall, cfi-icall.

Android CFI:
Google has active research on CFI (Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM).



< Software Defense />