University of BRISTOL

Software Defence mechanisms

Dr. Alma Oracevic
alma.oracevic@bristol.ac.uk

bristol.ac.uk

Topics

- Compiler and OS level protections
  - Stack canary
  - W⊕X
  - ASLR
- Control-flow Integrity

Stack RET BoF Cause

- We saw:
  - At CALL, return address is saved on the stack
  - Stack grows downward
  - Local buffers are allocated onto the stack
  - Return address is POPed into the EIP
  - EIP can point to anywhere in the memory!

Protection of saved RET
GCC approach

- **StackGuard protection**
- Different prologue
  - push a canary into the stack
  - it’s a constant 0x000aff0d.
- Different Epilogue:
  - checks the stack to see if the canary is still there unchanged

Protection of saved RET
GCC approach

- **StackGuard protection**
- Different prologue
  - push a canary into the stack
  - it’s a constant 0x000aff0d.
- Different Epilogue:
  - checks the stack to see if the canary is still there unchanged
  - if so, it keeps going with normal execution flow,

Protection of saved RET
GCC approach

- **StackGuard protection**
- Different prologue
  - push a canary into the stack
  - it's a constant 0x000aff0d.
- Different Epilogue:
  - checks the stack to see if the canary is still there unchanged
  - if so, it keeps going with normal execution flow,
  - if not it aborts

```
function_prologue:
    pushl   $0x000aff0d         // push canary into the stack
    pushl   %ebp                // save frame pointer
    mov     %esp,%ebp           // saves a copy of current %esp
    subl    $108, %esp          // space for local variables

function_epilogue:
    leave                               // standard epilogue
    cmpl    $0x000aff0d,(%esp)          // check canary
    jne     canary_changed
    addl    $4,%esp                     // remove canary from stack
    ret
```

Why Does it work?

- With strcpy() type functions:
  - the 0x00 will stop strcpy() copying further!!
  - Saved RET not changed.

Why Does it work?

- With strcpy() type functions:
  - the 0x00 will stop strcpy() copying further!!
  - Saved RET not changed.
- Similarly other white spaces terminate strings.
- Called terminator canaries

What are problems?

- local variables located after (top of) **buf** are not protected.
- The saved frame pointer EBP can be altered.

StackShield protection

- save return addresses in an alternate memory space named **retarray** (size 256).
- Two other global variables used:

StackShield protection

- save return addresses in an alternate memory space named **retarray** (size 256).
- Two other global variables used:
- **rettop**, initialized on startup, and is the address in memory where retarray ends

StackShield protection

- save return addresses in an alternate memory space named **retarray** (size 256).
- Two other global variables used:
- **rettop**, initialized on startup, and is the address in memory where retarray ends
- **retptr** is the address where the next clone is to be saved.

- Function Prologue:
  - return address is copied from the stack to **retarray** and **retptr** is incremented.
- Function Epilogue:
  - Retrive saved clone RET and check it with the RET from the stack.
  - Two strategies: replace OR Exit.

With newer gcc: New Stack layout

- Function parameters
- Function return address
- Frame pointer
- Cookie
- Locally declared variables and buffers
- Callee save registers

(Pro|epi)logue change

- set aside an additional 8 (4)- bytes.
- add extra instructions as follows:

```
869:   48 8b 4d f8              mov    -0x8(%rbp),%rcx
86d:   64 48 33 0c 25 28 00     xor    %fs:0x28,%rcx
874:   00 00
876:   74 05                    je     87d <main+0xf3>
878:   e8 c3 fd ff ff           callq  640 <__stack_chk_fail@plt>
87d:   c9                       leaveq
```

(Pro|epi)logue change

- set aside an additional 8 (4)-bytes.
- add extra instructions as follows:

```
000000000000078a <main>:
 78a:   55                      push   %rbp
 78b:   48 89 e5                mov    %rsp,%rbp
 78e:   48 83 ec 60             sub    $0x60,%rsp
 792:   89 7d ac                mov    %edi,-0x54(%rbp)
 795:   48 89 75 a0             mov    %rsi,-0x60(%rbp)
 799:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
 7a0:   00 00
 7a2:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 7a6:   31 c0                   xor    %eax,%eax
```

```
869:   48 8b 4d f8             mov    -0x8(%rbp),%rcx
86d:   64 48 33 0c 25 28 00    xor    %fs:0x28,%rcx
874:   00 00
876:   74 05                   je     87d <main+0xf3>
878:   e8 c3 fd ff ff          callq  640 <__stack_chk_fail@plt>
87d:   c9                      leaveq
```

W⊕X

- Write OR Execute
- Memory pages have permission to either writable or executable.
- Subclass of DEP
- Protects against attacks that rely on code execution on data segments.

Address Space Layout Randomization

- ASLR is implemented in Linux since kernel 2.6.12 (2005)
- Microsoft implemented ASLR in Windows Vista Beta 2 (2006)
- Originally ASLR was part of the PageEXec (PaX) project.

What is ASLR?

- With ASLR, the predictability of memory addresses is reduced by randomizing the address space layout for each instantiation of a program.
- Thus heap, stack, bss, text sections of the program get different addresses

ASLR in Linux

- There are different level of ASLR
  - `$ cat /proc/sys/kernel/randomize_va_space`
- *To set a level*
  - *`$echo N`* `|sudo tee /proc/sys/kernel/randomize_va_space`
  - *Where N is:*
    - *1 := No randomization*
    - *2 := Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized.*
    - *3:= Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized.*
  - *Binary should be compiled as position independent code.*

Extra details

- brk(2) Randomization (fs/binfmt_elf.c):

```
static int load_elf_binary(struct linux_binprm *bprm, struct pt_regs *regs)
{
        struct file *interpreter = NULL; /* to shut gcc up
        */ unsigned long load_addr = 0, load_bias = 0;
  ...
#ifdef arch_randomize_brk
        if ((current->flags & PF_RANDOMIZE) && (randomize_va_space > 1))
                current->mm->brk = current->mm->start_brk =
                        arch_randomize_brk(current->mm);
#endif
```

- Stack Randomization (arch/x86/kernel/process.c):

```
{
        if (!(current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
                sp -= get_random_int() % 8192;
        return sp & ~0xf;
}
```

References on ASLR

- Shacham et al. "On the effectiveness of address-space randomization" (CCS 2004).
- Optional:
  - PaX documentation (http://pax.grsecurity.net/docs/)

    Bhatkar, Sekar, DuVarney. "Efficient techniques for comprehensive protection from memory error exploits" (Usenix Security 2005).

Control Flow Integrity (CFI)

Control Flow Integrity (CFI)*

- A strong attack mitigation technique
- It *restricts* the control-flow of an application to *valid* execution traces.
- Assumption is that any attack deviates from the intended (*valid*) execution.
- Valid execution → precomputed (static) states
- At runtime, If an invalid state is detected, an alert is raised, usually terminating the application.

*Based on articles by Prof. Mathias Payer, EPFL

How it is different from bug detection

- Other runtime monitoring techniques, like sanitizers (ASan, UBSan, etc., available in compilers) target development settings, detecting violations when testing the program (e.g. during fuzzing).
- CFI, on the other hand, is an active (runtime) defence mechanism to mitigate an ongoing attack.

CFI Mechanism

- CFI detects control-flow hijacking attacks by limiting the targets of control-flow transfers.
- It consists of two abstract components:
  - A1: Static analysis component that recovers the Control-Flow Graph (CFG) of the application (*at different levels of precision*);
  - A2: the dynamic enforcement mechanism that restricts control flows according to the generated CFG. It involves instrumentation (as we discussed earlier under dynamic analysis lecture)

Example

```
void bar();
void baz();
void buz();
void bez(int, int);

void foo(int usr) {
  void (*func)();
//func either points to bar or baz
  if (usr == 0xaddaabba)
    func = bar;
  else
    func = baz;
  Func();
  return 0;
}
```

Example

```
void bar();
void baz();
void buz();
void bez(int, int);

void foo(int usr) {
  void (*func)();
//func either points to bar or baz
  if (usr == 0xaddaabba)
    func = bar;
  else
    func = baz;
  Func();
  return 0;
}
```

Example

```
void bar();
void baz();
void buz();
void bez(int, int);

void foo(int usr) {
  void (*func)();
//func either points to bar or baz
  if (usr == 0xaddaabba)
    func = bar;
  else
    func = baz;
  Func();
  return 0;
}
```

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %edi,-0x14(%rbp)
cmpl   $0xaddaabba,-0x14(%rbp)
jne    67d <foo+0x21>
lea    -0x3d(%rip),%rax
mov    %rax,-0x8(%rbp)
jmp    688 <foo+0x2c>
lea    -0x43(%rip),%rax
mov    %rax,-0x8(%rbp)
mov    -0x8(%rbp),%rdx
mov    $0x0,%eax
callq  *%rdx
nop
leaveq
retq
```

Example

```
void bar();
void baz();
void buz();
void bez(int, int);

void foo(int usr) {
  void (*func)();
//func either points to bar or baz
  if (usr == 0xaddaabba)
    func = bar;
  else
    func = baz;
  Func();
  return 0;
}
```

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %edi,-0x14(%rbp)
cmpl   $0xaddaabba,-0x14(%rbp)
jne    67d <foo+0x21>
lea    -0x3d(%rip),%rax
mov    %rax,-0x8(%rbp)
jmp    688 <foo+0x2c>
lea    -0x43(%rip),%rax
mov    %rax,-0x8(%rbp)
mov    -0x8(%rbp),%rdx
mov    $0x0,%eax
callq  *%rdx
nop
leaveq
retq
```

C++ virtual func calls

Controlling control-flow

- Control-flow can be categorized in two ways
  - unconditional and conditional jumps
  - Direct (target is known at compile time) and indirect (target is known at runtime)
- Due to write integrity of code section, direct transfers are protected
- It is the indirect control-flow that we want to protect.

Types of indirect control flows

- Forward-edge transfers: direct code forward to a new location and are used in indirect jump and indirect call instructions. e.g. `jmp *rax` and `callq *rbx`.
- Barkward-edge transfers: used to return to a location that was used in a forward-edge earlier, e.g., when returning from a function call through a return instruction.

Types of indirect control flows

- Forward-edge transfers: direct code forward to a new location and are used in indirect jump and indirect call instructions. e.g. `jmp *rax` and `callq *rbx`.
- Barkward-edge transfers: used to return to a location that was used in a forward-edge earlier, e.g., when returning from a function call through a return instruction.
- Accordingly, we have forward- and backward-edge CFI

Constructing CFG (A1)

- Depending on the precision, a combination of static and dynamic CFG construction can be used.
- Forward-edge over-approximates the targets of the indirect transfer.
- Indirect function calls are complex:
  - Approx. Based on function prototypes
  - Different CFI mechanisms use different forms of type equality, e.g., any valid function, functions with the same arity (number of arguments), or functions with the same signature (arity and equivalence of argument types). At runtime, any function with matching signature is allowed.

- Function matching can further be enhanced by looking at **address-taken** functions
  - Functions whose address is calculated and assigned.
- There are more complex approaches to enhance the precision, e.g. path sensitive computations.

Runtime enforcement (A2)

Runtime enforcement (A2)

- For forward-edge transfers, the code is often instrumented with some form of equivalence check-- only valid targets are allowed.

Runtime enforcement (A2)

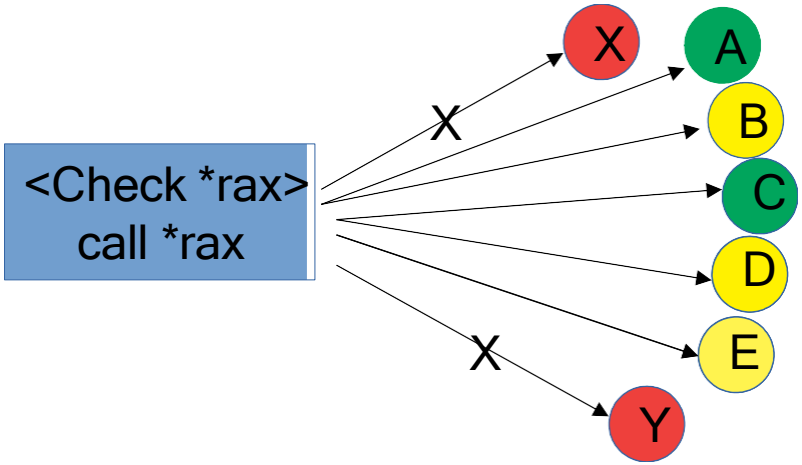
- For forward-edge transfers, the code is often instrumented with some form of equivalence check-- only valid targets are allowed.
- Each callsite and function entry-point is instrumented to check this equivalence, e.g. trampoline code.

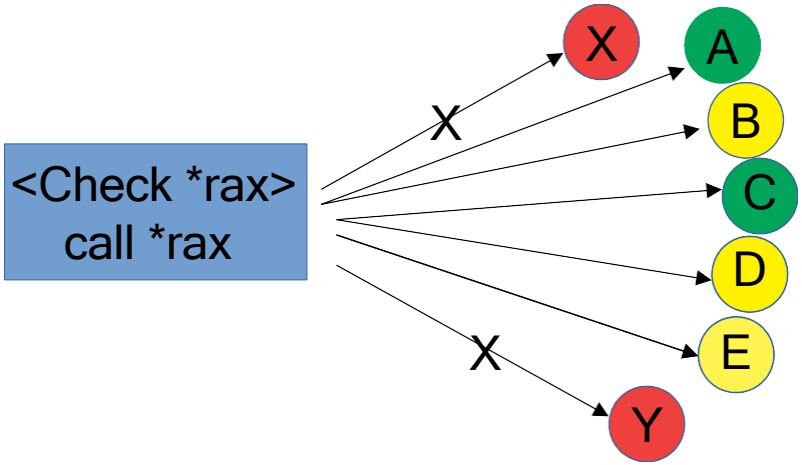
Runtime enforcement (A2)

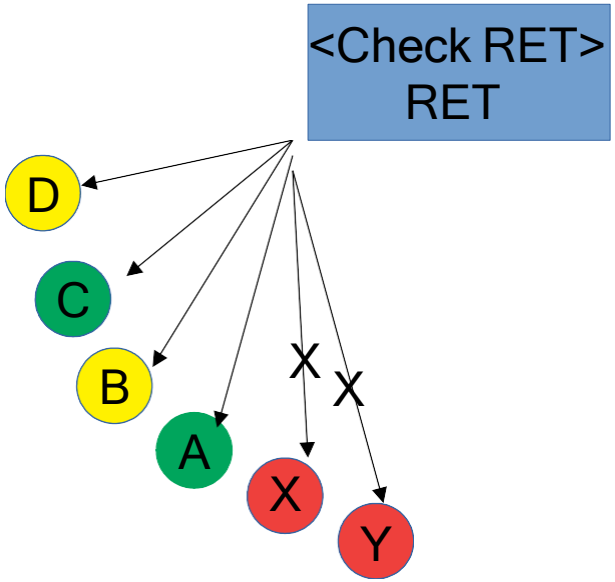
- For forward-edge transfers, the code is often instrumented with some form of equivalence check-- only valid targets are allowed.
- Each callsite and function entry-point is instrumented to check this equivalence, e.g. trampoline code.
- Backward-edge transfers are harder to model as returns can come from any valid callsite.

Runtime enforcement (A2)

- For forward-edge transfers, the code is often instrumented with some form of equivalence check-- only valid targets are allowed.
- Each callsite and function entry-point is instrumented to check this equivalence, e.g. trampoline code.
- Backward-edge transfers are harder to model as returns can come from any valid callsite.
- Shadow-stacks are used to enforce the recent caller.


<Check *rax>
call *rax
X
X
X
A
B
C
D
E
Y


<Check *rax>
call *rax
X
X
A
B
C
D
E
X
Y


<Check RET>
RET
D
C
B
A
X
X
X
Y

Example revisit

```
void  bar();
void  baz();
void buz();
void  bez(int,  int);
void foo(int usr) {
void (*func)();
  // func either points to bar or baz
  if (usr == 0xaddaabba)
    func = bar;
  else
    func = baz;

  // forward edge CFI check
  // depending on the precision of CFI:
  // a) all functions {bar, baz, buz, bez, foo} are allowed
  // b) all functions with prototype "void (*)()" are allowed,
  //    i.e., {bar, baz, buz}
  // c) only address taken functions are allowed, i.e., {bar, baz}
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Current state of CFI

- There are several academic proposals for CFI with varying degree of precision and performance.
- Mainstream systems have started shipping with some form of CFI
  - Intel CET (control enforcement technology)
  - Microsoft's Hardware-enforced Stack Protection
  - Clang/LLVM compiler