CS590-SWS/527 Software Security

Defense Mechanisms

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Kyriakos Ispoglou

`https://nebelwelt.net/teaching/16-527-SoftSec/`

Spring 2016

A model for Control-Flow Hijack attacks

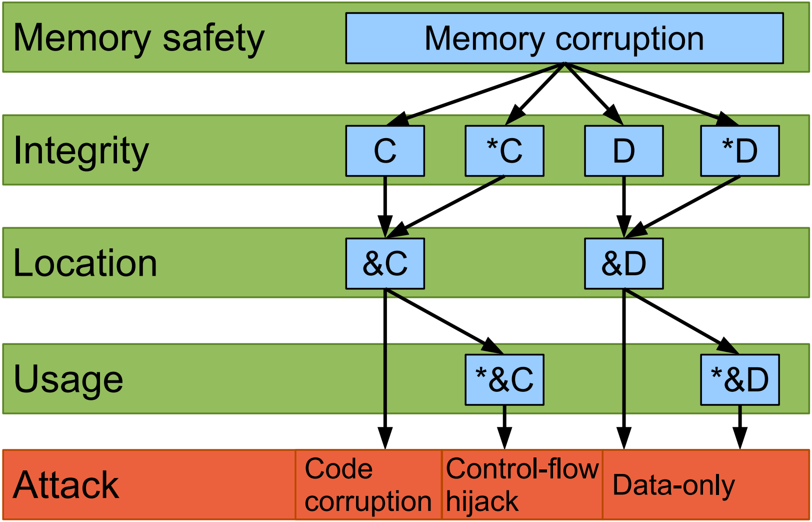


Table of Contents

1. Widely-adopted defense mechanisms
2. Data Execution Prevention
3. Address Space Randomization
4. Stack Canaries
5. Safe Exception Handling
6. Fortify source
7. Summary and conclusion

Widely-adopted defense mechanisms

- Hundreds of defense mechanisms were proposed.
- Only few defense mechanisms have been adapted in practice.
- What increases the chances of adaption?
- Mitigation of the most imminent problem.
- (Very) low performance overhead.
- Fits into the development cycle.

How not to get your defense mechanism adapted

- Require source code changes.
- Have complicated dependencies.
- Result in huge slowdown or have particularly bad worst-case outliers.
- Patent your defense mechanism and try to sell it.

Table of Contents

1. Widely-adopted defense mechanisms
2. Data Execution Prevention
3. Address Space Randomization
4. Stack Canaries
5. Safe Exception Handling
6. Fortify source
7. Summary and conclusion

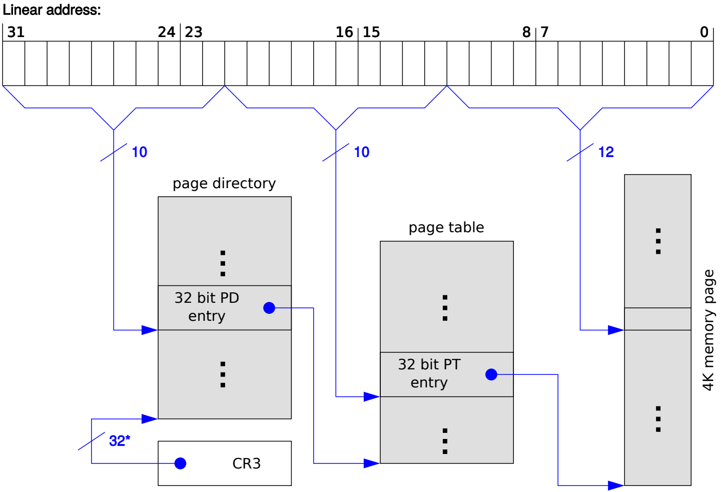
Data Execution Prevention (DEP)

- Historically, the x86 ISA did not distinguish between code and data.
- Any data in the process could be interpreted as code.
- The attack succeeds if an attacker can redirect the control flow to attacker controlled data.
- Assumption: if an attacker cannot inject code (as data), then a code execution attack is not possible.

DEP

- Intel and AMD extended page tables and introduced the NX-bit (No eXecute bit).
- Thanks to the joy of marketing, Intel calls this bit XD (eXecute Disable), AMD calls it Enhanced Virus Protection, and ARM calls it XN (eXecute Never).
- This is an additional bit for every mapped virtual page. If the bit is set, then data on that page cannot be interpreted as code and the processor will trap if control flow reaches that page.

“Old-school” Page Tables



*) 32 bits aligned to a 4-KByte boundary

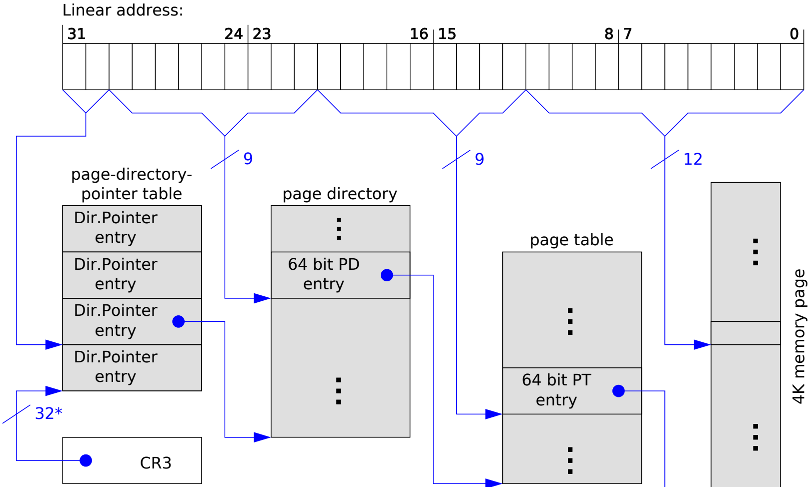
(c) RokerHRO, Wikimedia.

“Old-school” Page Table Entry

Bit Position	Content
0 (P)	Present; must be 1 to map a 4-KB page
1 (R/W)	Read/write; if 0, writes may not be allowed
2 (U/S)	User/supervisor; if 0 access with CPL=3 not allowed
3 (PWT)	Page-level write-through
4 (PCD)	Page-level cache disable
5 (A)	Accessed
6 (D)	Dirty
7 (PAT)	Page Attribute Table (PAT) indicator or reserved (0)
8 (G)	Global
11:9	Ignored
31:12	Physical address of the 4-KB page

According to Intel 64 and IA-32 manual 3A, Table 4-6.

Physical Address Extension (PAE) Page Tables



*) 32 bits aligned to a 32-Byte boundary

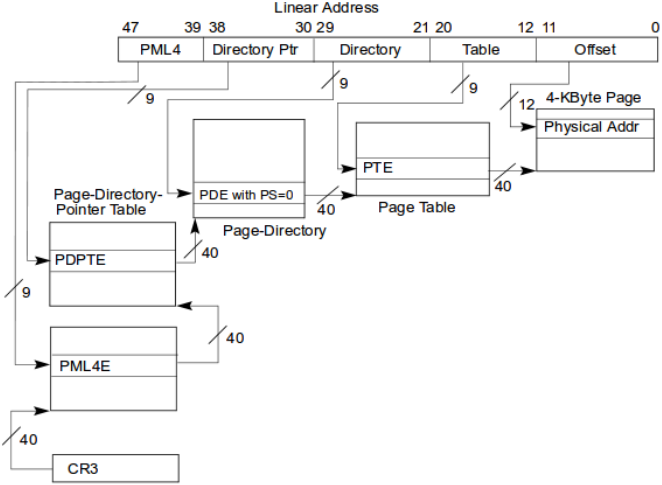
(c) RokerHRO, Wikimedia.

PAE Page Table Entry

Bit Position	Content
0 (P)	Present; must be 1 to map a 4-KB page
1 (R/W)	Read/write; if 0, writes may not be allowed
2 (U/S)	User/supervisor; if 0 access with CPL=3 not allowed
3 (PWT)	Page-level write-through
4 (PCD)	Page-level cache disable
5 (A)	Accessed
6 (D)	Dirty
7 (PAT)	Page Attribute Table (PAT) indicator or reserved (0)
8 (G)	Global
11:9	Ignored
(M-1):12	Physical address of the 4-KB page
62:M	Reserved (0)
63 (XD)	*If IA32_EFER.NXE = 1, execute-disable if 1*

According to Intel 64 and IA-32 manual 3A, Table 4-11.

x86-64 Page Tables



(c) Intel 64 and IA-32 manual 3A, Table 4-8.

x86-64 Page Table Entry

Bit Position	Content
0 (P)	Present; must be 1 to map a 4-KB page
1 (R/W)	Read/write; if 0, writes may not be allowed
2 (U/S)	User/supervisor; if 0 access with CPL=3 not allowed
3 (PWT)	Page-level write-through
4 (PCD)	Page-level cache disable
5 (A)	Accessed
6 (D)	Dirty
7 (PAT)	Page Attribute Table (PAT) indicator or reserved (0)
8 (G)	Global
11:9	Ignored
(M-1):12	Physical address of the 4-KB page
51:M	Reserved (0)
62:52	Ignored
63 (XD)	*If IA32_EFER.NXE = 1, execute-disable if 1*

Alternate approaches

- Not all hardware supports PAE.
- ExecShield by Ingo Molnar used code segment restrictions to limit code execution on x86 for Linux.
- OpenBSD's W^X follows the same idea.
- PaX, also for Linux, is ExecShield on steroids with significant remapping of code region (and may break applications).

Table of Contents

1. Widely-adopted defense mechanisms
2. Data Execution Prevention
3. Address Space Randomization
4. Stack Canaries
5. Safe Exception Handling
6. Fortify source
7. Summary and conclusion

Address Space Randomization (ASR)

- Successful control-flow hijack attacks depend on the attacker overwriting a code pointer with a known alternate target.
- ASR changes (randomizes) the process memory layout.
- If the attacker does not know where a piece of code (or data) is, then it cannot be reused in an attack.
- Attacker must first “learn” or recover the address layout.

Challenges for ASR

- Information leakage (e.g., through side channels)
- Brute forcing a secret value
- Rerandomization (for “long” running processes)

ASLR effectiveness

ASLR effectiveness

The security improvement of ASLR depends on (i) the entropy available for each randomized location, (ii) the completeness of randomization (i.e., are all objects randomized), and (iii) the lack of any information leaks.

Candidates for randomization

- Trade-off between overhead, complexity, and security benefit of randomization.
- Randomize start of heap
- Randomize start of stack
- Randomize start of code (for executable – PIE and for each library – PIC)
- Randomize mmap allocated regions
- Randomize individual allocations (malloc)
- Randomize the code itself, e.g., gap between functions, order of functions, basic blocks
- Randomize members of structs, e.g., padding, order

Related concept: software diversity

ASLR entropy

- Assuming all objects are randomized, entropy of each section is key to security.
- Attacker will follow path of least resistance, i.e., target the object with the lowest entropy.
- Early ASLR implementations had low entropy on the stack and no entropy on x86 for the main executable.
- Linux (through Exec Shield) uses 19 bits of entropy for the stack (on 16 byte period) and 8 bits of mmap entropy (on 4096 byte period).

Table of Contents

Stack canaries

- Most attacks in the early 2000s relied on a stack-based buffer overflow to inject code.
- Memory safety would mitigate this problem but adding full safety checks is not feasible due to high performance overhead.
- Instead of checking each dereference to detect arbitrary buffer overflows we can add a check that checks the integrity of a certain variable.
- Assume we only want to prevent control-flow hijack attacks.
- We therefore only need to protect the integrity of the return instruction pointer.
- Buffer overflows only happen if pointer arithmetic is involved.

Stack canaries

- Place a canary after a *potentially* vulnerable buffer and before the return instruction pointer.
- Check the integrity of the canary before the function returns.
- The compiler may place all buffers at the end of the stack frame and the canary just before the first buffer. This way, all non-buffer local variables are protected as well.
- Disadvantage: the stack canary only protects against continuous overwrites iff the attacker does not know the canary.
- Iff the attacker knows the secret or the attacker uses a direct overwrite then the defense is not effective.
- An alternative is to encrypt the return instruction pointer by xoring it with a secret.

Stack overflow – no stack protector

```c
char unsafe(char *vuln) {
  char foo[12];
  strcpy(foo, vuln);
  return foo[1];
}

int main(int ac,
    char* av[]) {
  unsafe(argv[0]);
  return 0;
}
```

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %rdi,-0x18(%rbp)
mov     -0x18(%rbp),%rdx
lea     -0x10(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   400410 <strcpy@plt>
movzbl  -0xf(%rbp),%eax
leaveq
retq
```

Stack overflow – with stack protector

```
push   %rbp;mov %rsp,%rbp
sub     $0x20,%rsp
mov     %rdi,-0x18(%rbp)



mov     -0x18(%rbp),%rdx
lea     -0x10(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   <strcpy@plt>
movzbl  -0xf(%rbp),%eax




leaveq; retq
```

```
push    %rbp; mov %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     %fs:0x28,%rax
mov     %rax,-0x8(%rbp)
xor     %eax,%eax
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   <strcpy@plt>
movzbl  -0x1f(%rbp),%eax
mov     -0x8(%rbp),%rcx
xor     %fs:0x28,%rcx
je      <unsafe+0x46>
callq <__stack_chk_fail@plt>
leaveq; retq
```

Table of Contents

1. Widely-adopted defense mechanisms
2. Data Execution Prevention
3. Address Space Randomization
4. Stack Canaries
5. Safe Exception Handling
6. Fortify source
7. Summary and conclusion

Exceptions in C++

```
double div(double a, double b) {
   if (b == 0)
      throw "Division by zero!";
   return (a/b);
}
...
try {
  result = div(foo, bar);
} catch (const char* msg) {
  ...
}
```

- Exceptions allow handling of special conditions.
- Exception-safe code safely recovers from thrown conditions.
- A “safe” alternative to ugly goto statements.

Exception implementation

- Exception handling requires support from the code generator (compiler) and the runtime system (libc or libc++).
- There are two fundamental approaches: (a) inline exception information in stack frame or (b) generate exception tables that are used when an exception is thrown.

Inline exception handling

- The compiler generates code that registers exceptions whenever a function is entered.
- Individual exception frames are linked across stack frames.
- When an exception is thrown, the runtime system follows the chain of exception frames to find the corresponding handler to a specific exception.
- This approach is compact but results in some overhead for each function call (as metadata about exceptions has to be allocated).

Exception tables

- Code generation also emits tables that relate ranges of instruction pointers to program state with respect to exception handling.
- Throwing an exception is tranlsated into a range query in these table that determines the correct handler for the exception.
- These tables are encoded very efficiently. This encoding may lead to security problems.

Windows exceptions

Microsoft Windows uses a combination of tables and inlined exception handling. Each stack frame records (i) unwinding information, (ii) the set of destructors that need to run, and (iii) the exception handlers if a specific exception is thrown.

When entering a function, a structured exception handling (SEH) record is generated, pointing to a table with address ranges for try-catch blocks and destructors. Handlers are kept in a linked list:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
   struct _EXCEPTION_REGISTRATION_RECORD *Next;
   PEXCEPTION_ROUTINE                     Handler;
} EXCEPTION_REGISTRATION_RECORD,
  *PEXCEPTION_REGISTRATION_RECORD;
```

Attacking Windows exception handling

- An attacker may overwrite the first SEH record on the stack and point the handler to the first gadget.
- Microsoft developed two defenses: SAFESEH and SEHOP.
- SafeSEH forces the compiler to generate a list of allowed targets. If a record points to an unknown target it is rejected.
- SEHOP initializes the chain of registration records with a sentinel. If the sentinel is not present, the handler is not executed.
- How strong are these defenses? Discuss.

GCC exception handling

- GCC encodes all exception information in external tables.
- When an exception is thrown, the tables are consulted to learn which destructors need to run and what handlers are registered for the current IP location.
- This results in less overhead in the non-exception case (as additional code is only executed "on-demand" but otherwise jumped over.
- The information tables can become large and heavyweight "compression" is used, namely an interpreter that allows on-the-fly construction of the necessary data.
- This interpreter can be (ab-)used for Turing-complete execution[1].

[1] James Oakley and Sergey Bratus, Exploiting the Hard-Working DWARF, WOOT'11.

Table of Contents

1. Widely-adopted defense mechanisms
2. Data Execution Prevention
3. Address Space Randomization
4. Stack Canaries
5. Safe Exception Handling
6. **Fortify source**
7. Summary and conclusion

Format strings

- Format strings allow to generate formatted output.
- Format strings also allow to write to memory through `%n`.
- Format strings allow to jump over arguments and access stack slots out-of-bounds through, e.g., `%2$hn`.
- Format strings can be bad. Don't let the user control the first argument to `printf`.

Format string mitigations

- Deprecate use of `%n` (Windows).
- Add extra checks for all format strings (Linux):
  1. Check for buffer overflows if possible.
  2. Check if the first argument is in a read-only area.
  3. Check if all arguments are used.
- Linux checks the following functions: `mem{cpy,pcpy,move,set}`, `st{r,p,nc}py`, `str{,n}cat`, `{,v}s{,n}printf`.

Fortify source (buffers)

The GCC/GLIBC patch distinguishes between four cases:

1. Known correct: do not check.
2. Not known if correct, but checkable (i.e., compiler knows length of target): do check.
3. Known incorrect: compiler warning, do check.
4. Not known if correct, not checkable: no check, overflows may remain undetected.

Table of Contents

1. Widely-adopted defense mechanisms
2. Data Execution Prevention
3. Address Space Randomization
4. Stack Canaries
5. Safe Exception Handling
6. Fortify source
7. Summary and conclusion

Summary

- Several defense mechanisms have been adopted in practice. Know their strengths and weaknesses.
- Data Execution Prevenention stops code injection attacks, but does not stop code reuse attacks.
- Address Space Randomization is probabilistic, shuffles memory space, prone to information leaks.
- Stack Canaries is probabilistic, does not protect against direct overwrites, prone to information leaks.
- Safe Exception Handling protects exception handlers. Reuse remains possible.
- Fortify source protects static buffers and format strings.

Questions?

?