Software security, secure programming

Lecture 4: Protecting your code against software vulnerabilities ? (overview)

Master M2 Cybersecurity

Academic Year 2025 - 2026

Preamble

Bad news

several **(widely used !)** programming languages are unsecure . . .

- codes are likely to contain vulnerabilities
- some of them can be exploited by an attacker . . .

Good news

Ther exists some protections to make attacket's life harder !

$\rightarrow$ 3 categories of protections:

- from the programmer (and/or programming language) itself
- from the compiler / interpreter
- from the execution plateform

Outline

Programmer's level protections

Compilers level protections

Plateform level protections

Bonus

Step 0: all the languages are not equal . . .

2 main issues:

1. how much the compiler (and not the developer) is in charge of security ?
2. what about unsecure programs ?
   (exploitable vs (random) crash vs exception raised vs compiler rejected)

- **unsecure languages:** Assembly languages, C, C++
  weakly typed, side-effects, undefined behaviors, explicit pointers, explicit heap management, etc.
  ⇒ no memory safety, no type safety . . .
- **reasonably secure languages:** Java, C#, Ada, Python, Rust
  strongly typed, no pointers, garbage collector, ∼ type safety, but still some possible unsafe primitives/libraries
- **more secure languages ? :** OCaml, Haskell, Python (?), etc.
  strongly typed, no pointers, garbage collector, no side effects (immutable data)

→ **Of course:** trade-off between security, expressiveness, execution time, code re-use, etc.

Demo: C, Ada, Java

Step 1: Know the threats . . .

Most language level vulnerabilities are well-known !

CWE (Common Weakness Enumeration) https://cwe.mitre.org/

- a community-developed list of common software security weaknesses
- common language + a measuring stick for software security tools
- a baseline for weakness identification, mitigation, and prevention efforts

Ex: CWE-131 (Incorrect Calculation of Buffer Size)

CVE (Common Vulnerabilities and Exposures) https://cve.mitre.org/

An (exhaustive ?) open list of all the publicly known soft. vulnerabilities
→ provides a common name & a standardized description
Ex: CVE-2017-12705 (A Heap-Based Buffer Overflow in Advantech WebOP).

CAPEC (Common Attack Pattern Enumeration and Classification)

https://capec.mitre.org/

"A comprehensive dictionary and classification taxonomy of known attacks"
Attack scenario, the attacker perspective (means, gains), possible protections
→ a "design pattern" of an attack
Ex: CAPEC-100 (Overflow Buffers)

Step 2: and avoid the traps !

- The CERT coding standarts
  `https://www.securecoding.cert.org/`
  - covers several languages: C, C++, Java, etc.
  - rules + examples of non-compliant code + examples of solutions
  - undefined behaviors
  - etc.

- Microsoft banned function calls

- ANSSI recommendations
  - JavaSec, LaFoSec (Ocaml, F#, Scala)
  - Rules for Secure C language software

- Use of secure libraries
  - `Strsafe.h` (Microsoft)
    guarantee null-termination and bound to dest size
  - `libsafe.h` (GNU/Linux)
    no overflow beyond current stack frame
  - etc.

Etc. (a lot of available references about "secure coding" . . .)

CERT coding standarts - Example 1

INT30-C. Ensure that unsigned integer operations do not wrap

Example of non compliant code

```
void func(unsigned int ui_a, unsigned int ui_b) {
       unsigned int usum = ui_a + ui_b;
       /* ... */
}
```

Example of compliant code

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  if (usum < ui_a) {
    /* Handle error */
  }
  /* ... */
}
```

CERT coding standarts - Example 2

ARR30-C. Do not form or use out-of-bounds pointers or array subscripts

Example of non compliant code

```
char *init_block(size_t block_size, size_t offset,
                 char *data, size_t data_size) {
  char *buffer = malloc(block_size);
  memcpy(buffer + offset, data, data_size);
  return buffer;
```

Example of compliant code

```
char *init_block(size_t block_size, size_t offset,
                 char *data, size_t data_size) {
  char *buffer = malloc(block_size);
  if (NULL == buffer) { /* Handle error */ }
  if (data_size > block_size || block_size - data_size < offset) {
    /* Data won't fit in buffer, handle error */
  }
  memcpy(buffer + offset, data, data_size);
  return buffer;
}
```

Code validation

Several tools can also help to detect code vulnerabilities . . .

Dynamic code analysis

Instruments the code to detect runtime errors (beyond language semantics!)

- invalid memory access (buffer overflow, use-after-free)
- uninitialized variables
- etc.

$\Rightarrow$ No false positive, but runtime overhead ($\sim$ testing)
Tool examples: Purify, Valgrind, AddressSanitizer, etc.

Static code analysis

Infer some (over)-approximation of the program behaviour

- uninitialized variables
- value analysis (e.g., array access out of bounds)
- pointer aliasing
- etc.

$\Rightarrow$ No false negative, but sometimes “inconclusive” . . .
Tool examples: Frama-C, Polyspace, CodeSonar, Fortify, etc.

Dynamic analysis tool example: AddressSanitizer

Google, open-source plugin for clang/gcc (flag `-fsanitize=address`)

Targets

- buffer overflows (within stack, heap, or globals)
- use-after-free (heap), use-after-return (stack)
- memory leaks, ...

Means

- code instrumentation (load/store operations)
- use of redzones around variables memory area
- custom version of `malloc()`
  (redzone insertion, delay re-used of free memory, collect log information)

At work

- $\sim$ 2x slowdown (Valgrind is $\sim$ 20x) and 1.5x-3.x memory overhead
  ($\rightarrow$ ok for tests and/or fuzzing campaigns)
- # (security) bugs found in Chrome, Firefox, MySQL, gcc, etc.

Demo: AdSan

Static analysis example: Frama-C RTE

runtime error annotation pluging for the Frama-C plateform [CEA List]

Targets

potential runtime-errors and undefined behaviors

- invalid memory accesses
- arithmetic overflows on signed and unsigned integers
- invalid casts from float to int, etc.

Means

- **static** enhanced type checking ⇒ potential **false positives**
- lighweight optimizations (e.g., constant folding) to improve precision

At work

- exhibits potential RTE issues at the source level (assert annotations)
- to be **discharged** by hand and/or by other Frama-C plugins (Wp, Eva)

(see `https://frama-c.com/rte.html`)

Demo: Frama-C RTE

Outline

Programmer's level protections

Compilers level protections

Plateform level protections

Bonus

Compilers may help for code protection

Most compilers offer compilation options enforce security

Examples[1]

- stack protection
  - stack layout
  - FORTIFY (enforces the use of safe libraries, e.g., `__strcpy_chk`)
  - canaries (e.g, gcc stack protector)
  - shadow stack for return addresses
  - control-flow integrity (e.g., clang CFI, Java)
  - ...
- pointer protection
  - pointer encryption (PointGuard)
  - smart pointers (C++)
  - ...
- no "undefined behavior"
  e.g., enforce wrap around for unsigned int in C
  (`-fno-strict-overflow`, `-fwrapv`)
- etc.

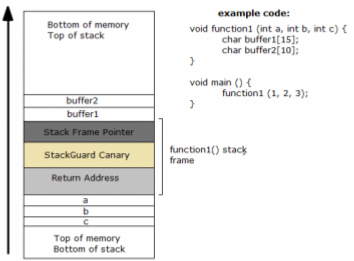
[1] see also `https://blog.quarkslab.com/clang-hardening-cheat-sheet.html` and E. Poll slides on the course web page)

Stack protection example: canaries

↪ gcc StackProtector, Redhat StackGuard, ProPolice, etc.

Principle: compiler generates extra code to:

1. insert a random value on the stack above the return address
2. check it before return and stops the execution if it has changed



Limited to **stack** (≠ heap) and **return** `@` (≠ loc. variables) protection

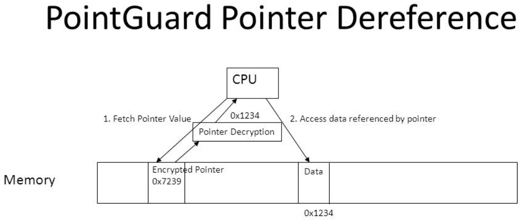
Possibly defeated by information disclosure, non consecutive overflow, etc.

`http://wiki.osdev.org/Stack_Smashing_Protector` Demo: -fstack-protector

Pointer protection

$\hookrightarrow$ Memory safety enforcement and attack prevention

- smart pointers: $\rightsquigarrow$ temporal memory safety
  ADT including pointer facilities + memory management (garbage collection)
  Ex: C++ template with unique/shared/weak pointers
- fat pointers: $\rightsquigarrow$ spatial memory safety
  extra meta-data to store memory cells base+bounds (**Ex:** C SoftBound)
- ciphered pointers: $\rightsquigarrow$ pointer integrity



Control-Flow Integrity (CFI)

The main idea

→ Ensure that the **actual** pgm control-flow is the one **intended** by the pgmer
several means:

- pre-compute all possible flows (CFG) and insert rutime-checks in the binary code
  pb: function pointers, dynamic calls (virtual functions), etc.
- simpler version: focus only on the call graph
  protect function calls and returns, possible over-approximations
- execution overhead: 20% - 40% ?

More details in Abadi et al. paper:
Control-Flow Integrity Principles, Implementations, and Applications
`https://users.soe.ucsc.edu/~abadi/Papers/cfi-tissec-revised.pdf`

CFI: software protection (e.g,. LLVM `clang`)

↪ focus on **Call Graph** . . .

Forward edges

↪ to enforce the validity of call statements
targets **virtual** and/or `indirect` function calls
Examples:

- C++ virtual functions, dynamic binding
- function pointers (`int *f(void)`)

check at runtime that the function type is the expected one

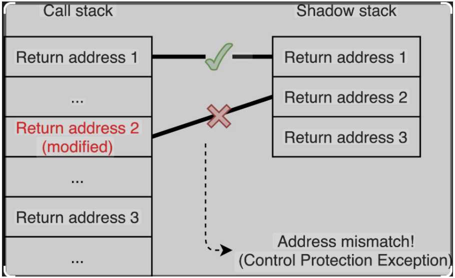
Backward edges

↪ to enforce the validity of return statements use a (software) shadow stack
to save a copy of return addresses
(located at random position, and protected against overflows)

see `https://blog.quarkslab.com/clang-hardening-cheat-sheet.html`

Shadow Stack

At call time:
↪ store function return adress in the (regular) call stack and in a copy (shadow stack)
At function termination:
↪ check that both stack pointers contains the same (return) addresses



Limitations:

- protect backward edges only
- the shadow stack should not be tampered with . . .

Bypassing CFI ? Use **data-oriented** attacks !

Principle:

↪ hijack code execution by modifying[2] non-control data (CFI preserving)

Attack objectives:

↪ information disclosure, priviledge escalation, security checks bypass

Existing techniques:

- Direct Data Manipulation: modify a single data directly through its address;
- Data/Block Oriented Programming (DOP/BOP): use a sequence of *"data-programming" gadgets/basic blocks* to chain several data corruptions

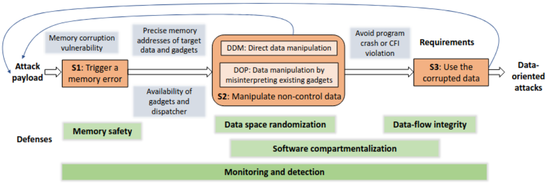


Fig. 2: Stages in data-oriented attacks and mitigation in different stages

Credits:

Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches

[2]unsing memory errors

Outline

Programmer's level protections

Compilers level protections

Plateform level protections

Bonus

Some more generic protections from the execution plateform

General purposes operating systems (Linux, Windows, etc.)

- Memory layout randomization (ASLR, KASLR)
  attacker needs to know precise memory addresses
  - make this address random (and changing at each execution)
  - no (easy) way to guess the current layout on a remote machine ...
  - randomized regions includes code, stack, heap and shared libraries
- Non executable memory zone (NX, W $\ominus$ X, DEP)
  basic attacks $\Rightarrow$ execute code from the data zone
  distinguish between:
  - memory for the code (eXecutable, not Writeable)
  - memory for the data (Writable, not eXecutable)

  Example: make the execution stack non executable ...

Rks:

- exists other dedicated protections for specific plateforms:
  e.g., JavaCard, Android, embedded systems, ...
- exists also hardware level protections:
  e.g., Intel SGC, ARM TrustZone, HW pointer protections, etc.

Defeating the ASLR ?

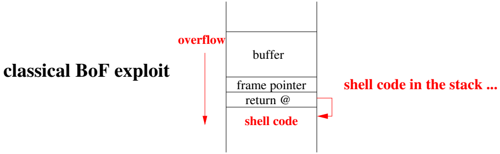
- some sections may not be be randomized (requires **Position Independent Executable**)
- On a 32 bits machine, brute force may be effective, e.g
  - heap spraying = filling the heap witth # copies of the payload
  - overwritting the LSB of a pointer
- Information leaks may help to fully disclose address information

$\Rightarrow$ needs to chain several (exploitable) vulns . . .

Stronger counter-measures ?

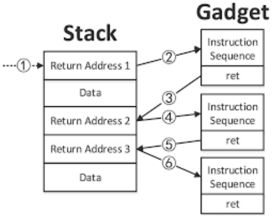
$\rightarrow$ encrypt the data stored in memory

Defeating the DEP ?



Do not store <u>shellcode</u> in the stack . . . use existing code instructions instead !

- <u>return-to-libc</u>: redirect the control-flow towards library code
- <u>return oriented programming</u> (ROP)
  payload = sequence of return-terminated instructions (<u>gadgets</u>)



- gadget programming is "turing complete"
- ∃ tools for gagdget extraction (ROPgadget, ROPium, etc.)
- ∃ ROP variants:
  COP (<u>call-oriented</u> programming), JOP (<u>jump-oriented</u> programming)

Rks: may also ∃ library calls allowing to make the stack executable . . .

Preventing ROP, COP, JOP ?

- preventing ROP:
  - count the number of `RET` instructions at runtime
  - use a **shadow stack** to duplicate return addresses
- preventing JOP and COP:
  use a new machine instruction to “tag” valid jump/call destinations
  e.g.: Intel CET (Control-Flow Enforcement Technology)

```
                ...
                CALL 0xabcdef
                ...
0xabcdef:
                ENDBRANCH  // tag a valid jum/call des
                ...
                RET
```

→ no (easy) way to jump in the middle of a function . . .

HW protection examples: CET and PAC

Intel CET (Control-flow Enforcement Technology)

Shadow stack (not readable/writable by softare)
+
Indirect branch tracking

uses `endbranch` label to mark legitimate branch targets, $\sim$ nop on old CPUs

ARM PAC (Pointer AuthentiCation)

- unused bit addresses in 64 bits architecture
  $\hookrightarrow$ can be used to store some pointer authentication value
  (assigned before writting in memory and verified before each use)
- new instructions to sign and authenticate
  cryptp algo = QUARMA, 128 bit key + some “context value”
- subsumes canaries (return address is protected), enforces CFI (indirect calls)
- Available on iOS

A comparison of three CFI protection techniques

LLVM Clang CFI vs Intel CET vs ARM PAC

Feature	LLVM CFI	Intel CET	ARM PAC
Type	Software (Compiler)	Hardware (CPU)	Hardware (CPU)
Protects	Indirect calls/jumps	Returns & indirect branches	Pointers (returns, vtables, etc)
Granularity	Type-based	Structure-based (IBT)	Cryptographic pointer integrity
Return Address Protection	No (needs SCS)	Yes (Shadow Stack)	Yes (PAC on LR register)
Performance Overhead	Low–Moderate	Very Low	Very Low
Hardware Required	No	Intel Tiger Lake+	ARMv8.3+
Platform Support	Cross-platform (LLVM)	x86 only	ARM64 only
Used In	Android, Chrome, Fuchsia	Windows 10+, Linux (opt-in)	iOS, macOS, Android (some SoCs)

Credits: A Deep Dive into Control Flow Protection

Outline

Programmer's level protections

Compilers level protections

Plateform level protections

Bonus

Bonus: retrieving the stack layout

Stack layout of the following code ?

```
int main() {
   int x;
   int T[10];
   char i;
   T[i]=x;
}
```

1. print variable addresses: need to re-compile, not reliable . . .
   `printf("%x",&x); printf("%x",&i);`
   `printf("%x",&(T[0]);`
2. use a debugger (ex: `gdb`): need to re-compile, not reliable . . .
   ↪ set a breakpoint (`b main`), execute (`run`), print addresses (`p &i`)
3. disassemble the executable code (`objdump -S`, idaPro, Ghidra, etc.)
   ↪get variable offset w.r.t frame pointer `rpb` on (x86_64)

Bonus: summary of memory-related exploits

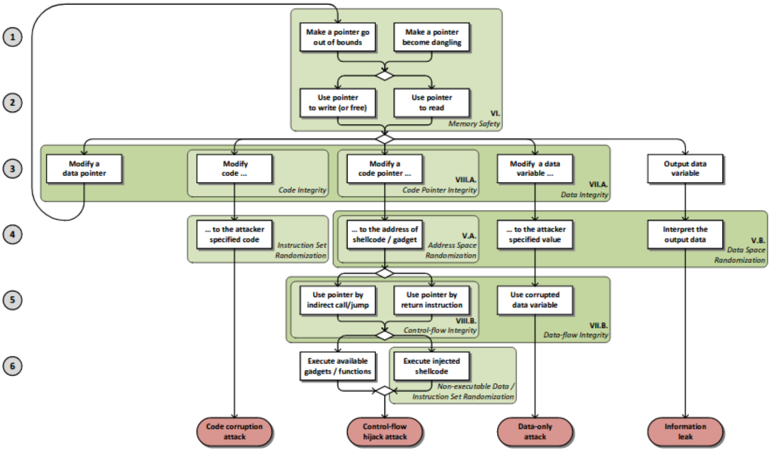


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

(from “SoK: Eternal War in Memory” Laszlo Szekeres et al., Oakland 13)

Some exploits defeating ASLR + DEP using ROP

CVE ID	Software	Vulnerability	Address leakage	User scripting
CVE-2011-0609	Adobe Flash	JIT type confusion	Read an IEEE-754 number	ActionScript
CVE-2012-0003	Windows Multimedia Library (affecting IE)	Heap buffer overflow	Read a string after overwriting its length	JavaScript
CVE-2011-4130	ProFTPD	Use-after-free	Overwrite the "226 Transfer Complete" message	none
CVE-2012-0469	Mozilla Firefox	Use-after-free	Read a string after overwriting its length	JavaScript
CVE-2012-1889	Microsoft Windows XML Core Services (affecting IE)	Uninitialized pointer	Read as a RGB color	JavaScript
CVE-2012-1876	Internet Explorer 9/10 (Pwn2Own 2012)	Heap buffer overflow	Read a string after overwriting its length	JavaScript

Table I
EXPLOITS THAT DEFEAT BOTH DEP AND ASLR USING ROP AND INFORMATION LEAKS

(from "SoK: Eternal War in Memory" Laszlo Szekeres et al., Oakland 13)

A more recent detailed example:

Exploiting CVE-2018-5093 on Firefox 56 and 57 (part 1 and part 2)

Conclusion

- $\exists$ numerous protections to avoid / mitigate vulnerability exploitations
- several protection levels
  code, verification tools, compilers, plateforms
- they allow to "(partially) mitigate" most known programming languages weaknesses (e.g., C/C++)
- they still require programmers skills and concerns
- even if they make attackers life harder . . .
- . . . they can still be bypassed !

$\rightarrow$ an endless game between "attackers" and "defenders" !