Software vulnerabilities

principles, exploitability, detection and mitigation

Laurent Mounier and Marie-Laure Potet

Verimag/Université Grenoble Alpes

GDR Sécurité – Cyber in Saclay (Winter School in CyberSecurity)

February 2021

Software vulnerabilities . . .

. . . are everywhere . . .

and keep going . . .

CVE
@CVEnew
CVE-2021-22305 There is a buffer overflow vulnerability in Mate 30 10.1.0.126(C00E125R5P3). A module does not verify the some input when dealing with messages. Attackers can exploit this vulnerability by sending malicious input through specific module. ... cve.mitre.org/cgi-bin/cvenam…

CVE
@CVEnew
CVE-2021-22304 There is a use after free vulnerability in Taurus-AL00A 10.0.0.1(C00E1R1P1). A module may refer to some memory after it has been freed while dealing with some messages. Attackers can exploit this vulnerability by sending specific message ... cve.mitre.org/cgi-bin/cvenam…

Outline

Software vulnerabilities (what & why ?)

Programming languages (security) issues

Exploiting a sofwtare vulnerability

Software vulnerabilities mitigation

Conclusion

Example 1: password authentication

Is this code “secure” ?

```
boolean verify (char[] input, char[] passwd , byte len) {
  // No more than triesLeft attempts
  if (triesLeft < 0) return false ; // no authentication
  // Main comparison
  for (short i=0; i <= len; i++)
    if (input[i] != passwd[i]) {
       triesLeft-- ;
       return false ;  // no authentication
    }
  // Comparison is successful
  triesLeft = maxTries ;
  return true ;  // authentication is successful
}
```

functional property:

$$\texttt{verify}(\texttt{input}, \texttt{passwd}, \texttt{len}) \Leftrightarrow \texttt{input}[0..\texttt{len}] = \texttt{passwd}[0..\texttt{len}]$$

What do we want to protect ? Against what ?

- confidentiality of `passwd`, *information leakage ?*
- control-flow integrity of the code
- no unexpected runtime behaviour, etc.

Example 2: `make 'python -c 'print "A"*5000'`

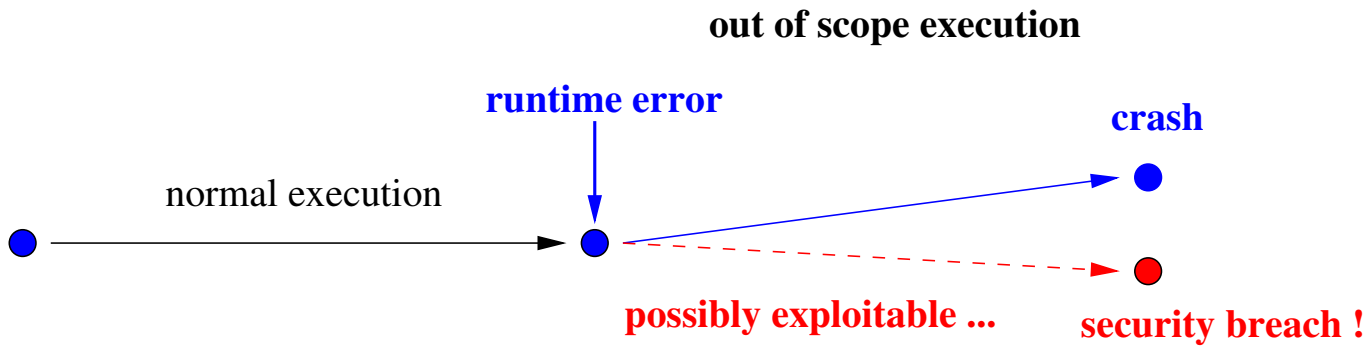
run `make` with a long argument $\rightsquigarrow$ **crash** (in recent Ubuntu versions)

Why do we need to bother about crashes (wrt. security) ?

crash = consequence of an unexpected run-time error
not trapped/foreseen by the programmer, nor by the compiler/interpreter

$\Rightarrow$ some part of the execution:

- may take place outside the program scope/semantics
- but can be controled/exploited by an attacker ($\sim$ ***"weird machine"***)



$\hookrightarrow$ may **break** all security properties ...
from simple denial-of-service to **arbitrary code execution**

Rk: may also happen **silently** (without any crash !)

Back to the context: computer system security

what does **security** mean ?

- a set of general **security** properties: CIA
  Confidentiality, Integrity, Availability (+ Non Repudiation + Anonymity + . . .)
- concerns the **running** software + the whole **execution plateform**
- depends on an intruder model
  $\rightarrow$ there is an "external actor" iœwith an **attack objective** in mind, and able to elaborate a dedicated strategy to achieve it ($\neq$ hazards)
  $\hookrightarrow$ something beyond **safety** and **fault-tolerance**

$\rightarrow$ **A possible definition:**

- functionnal properties = what the system should do
- security properties = what it should **not allow** w.r.t the intruder model . . .

The software security intruder spectrum

Who is the attacker ?

- external “observer”, black-box observations via side channels
  (execution time, power consumption)
- external user, interactions via regular input sources
  e.g., keyboard, network (man-in-the-middle), etc.
- another application running on the same plateform
  interacting through shared resources like caches, processor elements, etc.
- the execution plateform itself (e,g., when compromised !)

What is he/she able to do ?

At low level:

- unexpected memory read (data or code)
- unexpected memory write (data or code)

$\Rightarrow$ powerful enough for

- information disclosure
- unexpected/arbitrary code execution
- break code/data integrity

Two main classes of software vulnerabilites

Case 1 (not so common . . .)

Functional property not provided by a security-oriented component

- lack of encryption, too weak crypto-system
- bad processing of firewall rules
- etc.

Case 2 (**the vast majority !**)

Insecure coding practice in (any !) software component/application

- improper input validation $\rightsquigarrow$ SQL injection, XSS, etc.
- insecure shared resource management (file system, network)
- information leakage (lack of data encapsulation, side channels)
- exploitable run-time error
- etc.

But, why **software** security issues are so common ?

Software is **greatly involved** in “computer system security”:

- used to **enforce security properties**:
  crypto, authentication protocols, intrusion detection, firewall, etc.
- but also a major source of **security problems** . . .

Why ???

- we do not no very well how to write **secure** SW
  we do not even know how to write **correct** SW!
- behavioral properties can’t be validated on a (large) SW
  impossible by hand, untractable with a machine
- programmers feel not (so much) concerned with security
  time-to-market, security $\notin$ programming/SE courses
- heterogenous and nomad applications favor unsecure SW
  remote execution, mobile/embedded code, plugins, etc.
- widely used programming languages not designed for security
  most of them contain numerous traps and pitfalls

Outline

Defining a programming language

> *An unreliable programming language generating un- reliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.* *[C.A.R. Hoare]*

How to reduce this risk ?

Ideally, a programming language should:

→ avoid discrepancies between:

- what the programmer has in mind
- what the compiler/interpreter understands
- how the executable code **may** behave . . .

→ avoid program undefinedness and run-time errors . . .

→ provide well-defined **abstractions** of the execution plateform
. . . keep preserving expected expressivity level and runtime performances !

Types as a security safeguard ? (1)

"Well-typed programs never go wrong . . . " *[Robin Milner]*

Type safety

type safe language $\Rightarrow$ **NO** meaningless well-typed programs

$\hookrightarrow$ "out of semantic" programs are not executed, no **untrapped** run-time errors, no undefined behaviors, . . .

According to this definition:

- C, C++ are **not** type safe
- Java, C#, Python, etc. are type safe
  (when not using unsafe language constructs !)

Remarks about type safe languages:

- (meaningless) ill-typed programs can be rejected
  either at **compile time** or at **execution time**
- type systems are usually incomplete
  $\Rightarrow$ may also reject **meaningful** pgms (expressivity issue)

Undefined behaviors (in non type safe languages)

semantic "holes": meaningless source level compliant pgms . . .

Numerous ($\sim$ 190) undefined behaviors in C/C++

- **out-of-bound pointers:**
  (pointer + offset) should not go beyond object boundaries
- **strict pointer aliasing:**
  comparison between pointers to $\neq$ objects is undefined
- `NULL` pointer dereferencing
- **oversized shifts** (shifting more than n times an n-bits value)
- etc.

Compilers:

- may assume that undefined behaviors never happen
- have no "semantic obligation" in case of undefined behavior
  $\rightsquigarrow$ aggressive optimizations . . . able to **suppress security checks!**

$\Rightarrow$ **dangerous gaps** between pgmers intention and code produced . . .

Security issues with undefined behaviors [C]

Out-of-bounds buffer accesses are undefined

```
char i=0;
char t[10] ; // indexes may range from 0 to 9
t[10]=42;    // off-by-one buffer overflow
printf("%d\n", i) ;
```

What is the printed value ?

Signed integer overflows are undefined

```
int offset, len ;  // signed integers
...
if (offset < 0 || len <= 0)
    return -EINVAL;  // either offset or len is negative
// both offset and len are positive
if ((offset + len > INT_MAX ) || (offset + len < 0)
    return -EFBIG // offset + len does overflow
...
```

The `return -EFBIG` instruction may never execute . . . Why ?

Types as a security safeguard ? (2)

Weakly typed languages:

- implicit type cast/conversions
  integer $\rightsquigarrow$ float, string $\rightsquigarrow$ integer, etc.
- operator overloading
  - $+$ for addition between integers and/or floats
  - $+$ for string concatenation
  - etc.
- pointer arithmetic
- etc.

$\Rightarrow$ weaken type checking and may confuse the programmer . . .
(runtime type may not match with the intended operation performed)

In practice:

- happens in many widely used programming languages . . .
  (C, C++, PHP, JavaScript, etc.)
- may depend on compiler options / decisions
- often exacerbated by a lack of clear and un-ambiguous documentation

Rk: see also type confusion in OO-languages . . .

Possible problems with type conversions [bash]

```
PIN_CODE=1234
echo -n "4-digits pin code for autentication: "
read -s INPUT_CODE; echo

if [ "$PIN_CODE" -ne "$INPUT_CODE" ]; then
   echo "Invalid Pin code"; exit 1
else
   echo "Authentication OK"; exit 0
fi
```

There is a very simple way to pawn this authentication procedure . . .

Implicit type conversions [JavaScript] (2)

Array slicing with JavaScript

```
var a=[] ;
// fill array a with 100 values from 0.123 to 99.123
for (var i=0; i<100; i++) a.push(i + 0.123) ;
// fill array b with the 10 first values of a
var b = a.slice(0, 10);
```

$\rightsquigarrow$ b = [0.123, 1.123, 2.123, ..., 9.123]

Implicit conversion and object values

```
var c = a.slice(0, {valueOf:function (){return 10;}});
```

$\rightsquigarrow$ c = [0.123, 1.123, 2.123, ..., 9.123]

Now with an (un-detected) side effect . . .

```
var d = a.slice(0,valueOf:function(){a.length=0;return 10;}});
```

$\rightsquigarrow$ d = [0.123, 1.123, 2.1219959146e-313, 0, 0, ...]

$\rightarrow$ out-of-bounds read, memory leakage [CVE-2016-4622 in JavaScriptCore]

Memory safety (yet another highly desirable property !)

Only valid memory accesses should occur at runtime

valid ?

- of correct type and size $\rightsquigarrow$ no *spatial* memory violation
- properly allocated and initialized, "freshness" (no re-use)
  $\rightsquigarrow$ no *temporal* memory violation
- no memory leakage, etc.

Memory safety:

A pgm behavior should not **affect** *– nor* **be affected by** *–*
unreachable parts of its memory state

- formal definitions based on non-interference or separation logic
- some consensus:
  "*C (and C++) are* **not** *memory-safe*", "*Java (and C#) are considered memory-safe*", "*Rust is designed to be memory-safe*"
- real world context (finite memory space, unsafe language constructs) **weaken** memory safety in practice

Outline

How to “break” a software security as a regular user ?

$\rightarrow$ Some reminders about how a code executes at runtime . . .

At runtime:

- code + data = sequence of bits, with no physical distinction
  Ex: 000A7A33 $\rightsquigarrow$ `mov eax, ecx` or 686643 or `"DB+"` or . . .
- code + data are stored in the same (physical) memory

However, several ways to **hijack** the program control flow:

$\hookrightarrow$ numerous opportunities for a user to influence the code execution:

- read/write an unexpected data memory zone
  (may change a global/local variable, a parameter, etc.)
  $\rightarrow$ take an unexpected branch/while condition
- change the address or return value of a function called
- change the address of an exception handler
- etc.

Application 1: Stack-based buffer overflows

An old (but still effective !) way to drastically change a pgm control-flow . . .

Memory organization at runtime

- 3 main memory zones
  the code, the stack and the heap
  - heap : dynamic memory allocations
  - stack : function/procedures (dynamic) memory management
    local variables + parameters + temporaries + . . .
    + (current function) return addresses
- when a **write** access to a local variable with an **incorrect** stack address occurs it may overwrite stack data
- writting outside the bounds of an array is an example of such a situation

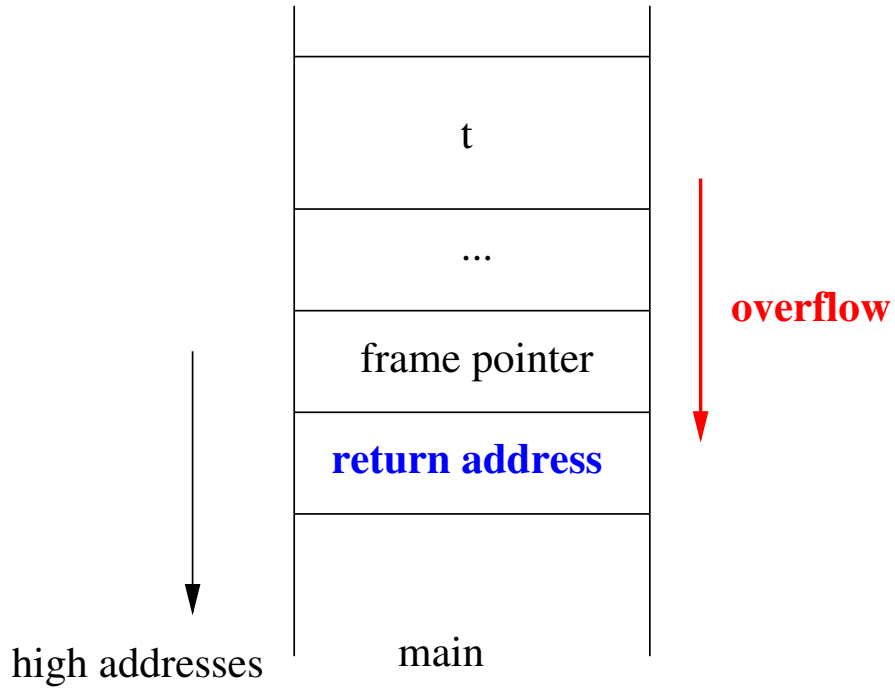
A “simple” recipe for cooking a buffer overflow exploit

1. find a pgm crash due to a controlable buffer overflow
2. fill the buffer s.t. the return address is overwritten with the address of a function you want to execute (e.g., a **shell command**)

Example

```
void f(char *s)
{
  char t[8] ;
  ...
  // copy s into t
  strcpy (t, s) ;
}

int main(...) {
    ...
   // user-controled arg
    f(argv[1]) ;
    ...
}
```



The `strcpy` function does not check for overflows

$\Rightarrow$

- the return address in the stack can be overwritten with a user input
- program execution can be fully controlled by a user . . .

Application 2: what about the heap ?

- a (finite) memory zone for dynamic allocations
- OS-level primitives for memory allocation and release
- various allocation/de-allocation strategies at the language level:
  - explicit allocation and de-allocation:

    ex: C, C++ (`malloc`/`new` and `free`)
  - explicit allocation + *garbage collection*:

    ex : Java, Ada (`new`)
  - implicit allocation + garbage collection:

    ex : CAML, JavaScript, Python

Example of (incorrect) heap memory use

```
void f (int a, int b)
{
  int *p1, *p2, *p3;
  p1 =( int *) malloc ( sizeof (int));  // allocation 1
  *p1 = a; // assigns *p1
  p2 = p1; // p2 is an alias of p1
  if (a > b)
        free (p1);  // p1 is now dangling !
  p3 = (int *) malloc (sizeof (int));  // allocation 2
  *p3 = b;  // assigns p3
  printf ("%d", *p2) ;
}
```

What is the printed value ?

Use-after-Free (definition)

Use-after-free on an execution trace

1. a memory block is allocated and assigned to a pointer `p`:
   `p = malloc(size)`
2. this bloc is freed later on: `free (p)`
   ↪ `p` (and all its aliases !) becomes a dangling pointer
   (it does not point anymore to a **valid** block)
3. `p` (or one of its aliases) is **dereferenced** . . . possibly without runtime error !

Vulnerable Use-after-Free on an execution trace

`p` points to a **valid block** when it is dereferenced (at step 3)
⇒ possible consequences:

- information leakage: `s = *p`
- write a sensible data: `*p = x`
- arbitrary code execution: `call *p`

Web navigators (IE, Firefox, Chrome, etc.) are favorite uaf targets !

Application 3: lack of input validation & sanitization

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

How to remove the whole filesystem using this PHP script ? `; rm -rf /`

Invalid/Unexpected program inputs $\rightsquigarrow$ 2 possible security flaws:

- Buggy parsing & processing *(input processing attack)*

  ex: invalid PDF file $\rightarrow$ buffer overflow $\rightarrow$ arbitrary code execution

  $\hookrightarrow$Incorrect input $\Rightarrow$ runtime error in the application . . .

- Flawed forwarding *(input injection attack)*

  ex: invalid web client input $\rightarrow$ SQL query to DB $\rightarrow$ info leakage

  $\hookrightarrow$Incorrect input $\Rightarrow$ forward an unsecure command to a back-end

  Untrusted facilities offered in many languages, e.g.:
  C/C++ (`system, execv`), Java (`Runtime.exec`), JavaScript (`eval`)

Bonus: a summary of memory-related exploits

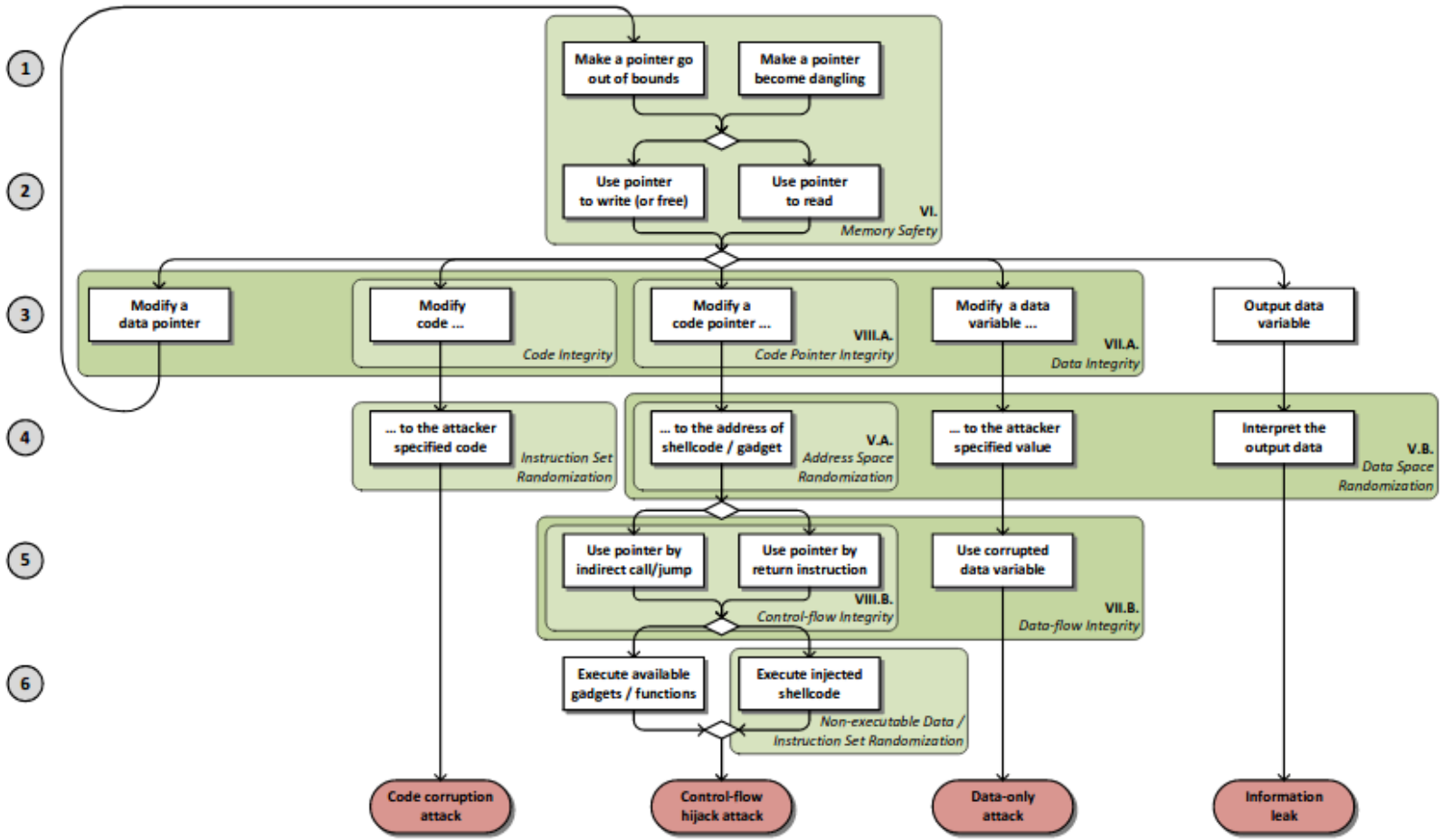


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

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

Outline

So far . . .

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

There 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

Knowing the traps . . . to avoid them !

- 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.

- 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.

And lots of valuable 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 */
  }
  /* ... */
}
```

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: Valgrind, AddressSanitizer, etc.

Static code analysis

Infer some (over)-approximation of the program behaviour

- 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.

. . . **and of course several classes of fuzzers**

Compilers may help for code protection

Most compilers offer compilation options enforce security

Examples

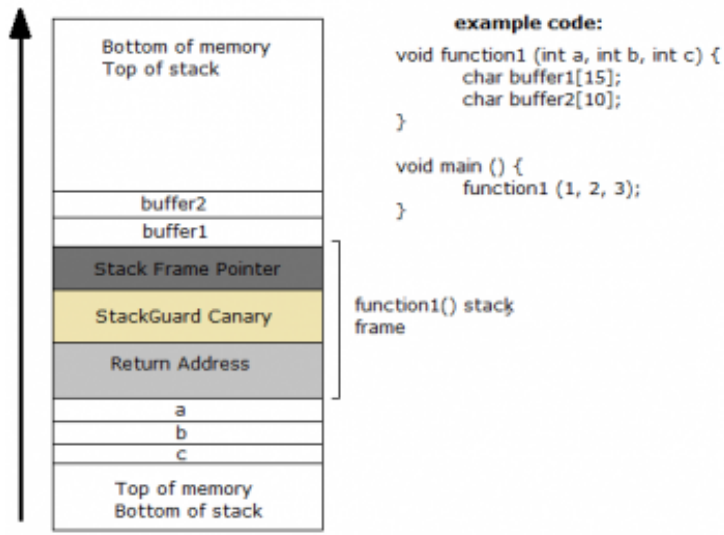
- stack protection
  - stack layout
  - canaries (e.g, gcc stack protector)
  - shadow stack for return addresses
  - ...
- pointer protection
  - pointer encryption (PointGuard)
  - smart pointers (C++)
  - ...
- no “undefined behavior”
  e.g., enforce *wrap-around* for signed int in C
  (`-fno-strict-overflow`, `-fwrapv`)
- etc.

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** ($\neq$ heap) and **return** @ ($\neq$ loc. variables) protection

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

Pointer protection

$\hookrightarrow$ Memory safety enforcement and attack prevention

- smart pointers: $\rightsquigarrow$ temporal memory safety

  Abstract Data Type 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

  HW mechanism for address encryption

  Ex: PointGuard

Control-Flow Integrity (CFI)

The main idea

$\rightarrow$ 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`

Clang CFI

Focus on virtual calls in C++ code
see `https://blog.quarkslab.com/clang-hardening-cheat-sheet.html`

Some more generic protections from the execution plateform

Memory layout randomization (ASLR)

the attacker needs to know memory addresses

- make this address random (and changing at each execution)
- no (easy) way to guess the current layout on a remote machine ...

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 ...

Outline

Future of software vulnerabilities ?

Some positive indications . . .

- Programming languages
  - how to choose a programming language ?
    mix from performance, knowledge, existing code, and security
  - new trends in programming language usage
    Python, Java, Rust (?) . . . but probably still many inappropriate C/C++ codes
- Security is gaining importance in software engineering cursus . . .
- More security oriented tools, compilers and execution plateforms
  $\rightarrow$ exploiting widely used SW becomes **quite hard**

$\hookrightarrow$ **towards a better control of the security/performance trade-off ?**

IEEE 2019 programming language ranking

Zerodium bug bounties

Some references

- “Mind your Language(s)” [Security & Privacy 2012]
  (E. Jaeger, O. Levillain, P. Chifflier - ANSSI)

- “Undefined Behavior: What Happened to My Code?” [APSys 2012]
  (X. Wang, H. Chen, A. Cheung, Z. Jia, M. Frans Kaashoek)

- “The Programming Languages Enthusiast” (Michael Hicks) blog
  - Software security is a programming language issue
  - what is type safety ?
  - what is memory safety ?

- E. Poll (Radboud University) web site

- etc. . . .