Software security, secure programming

Lecture 3: Programming languages (un)-security

Looking at the binary level

Master M2 Cybersecurity & MoSiG

Academic Year 2020 - 2021

Reminder

So far, we saw that:

- Unsecure softwares are (almost) everywhere . . .
- Programming languages (often) contribute to produce unsecure software:
  - misleading syntactic constructions
  - weak typing constraints, lack of type safety
  - undefined behaviors, lack of memory safety
  - etc.

$\Rightarrow$ **“source-level understanding”** $\neq$ **actual code behaviour**

But:

- how do this language weaknesses can be exploited at runtime ?
- what are typical intruder objective ?
- how can he/she operate ?

$\Rightarrow$ Let’s have a look at the **assembly code level** to answer . . .

Outline

- The intruder
- Arithmetic overflows
- Stack-based vulnerabilities
- Heap based vulnerabilities
- Type confusion vulnerabilities
- Input validation

The "software security" intruder

Intruder objectives

What can be expected when running an unsecure code ?

- break a CIA property, e.g.,
  read confidential data ; modify sensible data ;
  get priviledged accesses ; execute code of his own, etc.
- break application availability (Denial of Service), e.g., "hang up" a server
- (silently) hide/inject a malware (Non Repudiation)
- etc.

Intruder model

How can operate an intruder when running an unsecure code ?

As an external agent[1]: control program inputs & execution environment

Examples:

- fully control the keyboard, the network, the input files content, etc.
- partially control env. variables, file system, other process/threads
- cannot modify the code[2], break cryptography, etc.

[1] **other intruder models may also be considered . . . see later !**

[2] not always a valid assumption !

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

→ Some reminders about how a code executes at runtime . . .

At runtime:

- code + data = sequence of bits, with no physical distinction
  Ex: 000A7A33 ⇝ `mov eax, ecx` or 686643 or `"DB+"` or . . .
- code + data lie in the same (physical) memory
  - but usually in distinct zones
  - with some possible protections (e.g., “code zone cannot be over-written”)

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

→ numerous opportunities for a user to influence the code execution:

- take an unexpected branch condition
- read/write an unexpected data memory zone
  (may change a global/local variable, a parameter, etc.)
- change the address of a function called
- change the “return value" when a function terminates
- change the address of an exception handler
- etc.

(back to) Software vulnerabilities

An exploitable "bug", breaking some security property, w.r.t an intruder model

∃ several vulnerability taxonomies
(see `https://cwe.mitre.org/about/sources.html`)
Possible classification criteria:

- **unintended** (bug) vs **intentional** vulnerabilities (Trojan horse, backdoors, etc.)
- specification/source/binary level vulnerability
- location: application/operating system/hardware level
- etc.

∃ some international databases to record known software vulnerabilities

- Common Weaknesses Enumeration (CWE)
  classification of general known weaknesses
- Common Vulnerability Exposure (CVE)
  exhaustive (?) list of know vulnerability (for a given software)

∃ several secure coding standart
(w.r.t the programming language, application domain, intruder model, etc.)

Ex.: SEI CERT secure coding, MISRA, OWASP, etc.

Outline

Example 1: arithmetic overflows

Coding integers in base 2, on *n* bits

- signed integers: $[2^{n-1}, 2^{n-1} - 1]$; unsigned integers: $[0, 2^n - 1]$
- arithmetic operations:
  - possible overflow . . .
  - in case of overflow:
    either exception raised or wrap-around (mod *n*), or undefined
- signed $\leftrightarrow$ unsigned conversions:
  either forbidden, or explicitely / implicitely authorized
- conversions between several integer sizes:
  either forbidden, or explicitely / implicitely authorized

Example in C: if `x+y` overflows then

- "undefined behaviour" if signed
- wrap-around if unsigned . . .
- and if `x` signed and `y` unsigned ???

wrap-around + undefined behavior + implicit conversions = a dangerous coktail!

See rules 4 and 5 of the CERT Secure Coding Standard

Application to control-flow hijacking

```
unsigned int x ; // 32-bits unsigned integer
read (x) ;
if (x+1<10) {
// assume x < 9
// allocate x resources ...
} else {
    // assume x >= 9
}
```

$\rightarrow$ the “then” branch can be taken with $x = 2^{n-1}$ ...

```
signed int x=-1 ;           // 32-bits signed integer
unsigned int y=1;           // 32-bits unsigned integer
if (x<y) {
   ...
} else {
     // this should never happen ...
 ...
}
```

$\rightarrow$ the “else” branch is always taken !
(−1 being converted into a large signed value ...)

Outline

Example 2: stack-based buffer overflows

From "Smashing the stack for fun and profit" (Aleph One- 1996) to HeartBleed (2015) . . .

A historic (but still effective) way to drastically change a pgm control-flow . . .

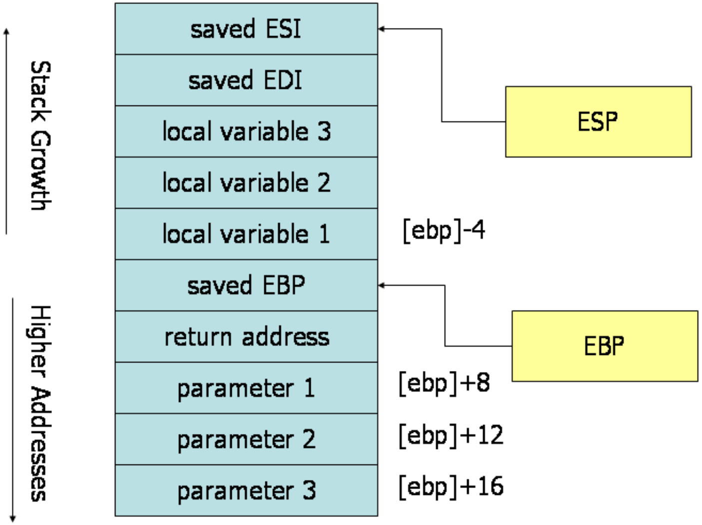
Memory organization at runtime

- 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 + . . .
    + 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 (unless runtime checks are inserted by the compiler . . .)

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**)

Stack layout for the x86 32-bits architecture



http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Application to control-flow hijacking (1)

```
void main ()
{
  char t;
  char t1[8] ;
  char t2[16] ;
  int i;
  t = 0;
  for (i=0;i<15;i++) t2[i]=2;
  t2[15]='\0' ;
  strcpy(t1, t2) ; // copy t2 into t1
  printf("the value of t is: %d \n", t);
}
```

- prints 2 as the value of `t` . . .
- if we increase the size of `t2` we get a crash . . .

Rks: the results obtained may depend on the compiler . . .

- ordering of the local variables in the stack
- buffer overflow protections enabled/disabled by default (e.g., `gcc -fstack-protector ...`)

Application to control-flow hijacking (2)

```
int f ()
{
  char x[256];
  char t1[8] ;
  int i;
  scanf("%s", x) ; // read a string into x
  strcpy (t1, x) ; // copy buffer x into buffer t1
  return 0 ;
}

int main {
    ...
f() ;
...
}
```

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 . . .

see next lectures !

Some variants on the same theme ...

Several stack elements direct the pgm control-flow:

- function return addresses
- pointers to functions
- addresses of objects methods (method tables)
- addresses of exception handlers
- etc.

All of them might be overwritten by user-controlled write operations, e.g.,

- using a buffer overflow to overwrite these locations
- overwritting a pointer to the stack
- overwritting an object
- etc.

See rules 6, 7 and 8 of CERT C secure coding standard

Outline

What about the heap ?

From the user point of view:

- a (finite) memory zone for dynamic allocations
- OS-level primitives for memory allocation and release
- 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

  $\rightarrow$ numerous allocation/de-allocation strategies . . .

At runtime, the heap can be viewed as:

- a set of disjoints memory blocks
- each block is either allocated or free (not both !)
- an allocated block contain user data + meta-data
- meta-data are used to retrieve the underlying heap structure, e.g., block sizes, set(s) of free blocks, etc.

Example of (incorrect) heap memory managememt

```
void f (int a, int b)
{
  int *p1, *p2, *p3;
  p1 =( int *) malloc ( sizeof (int));  // allocation 1
  *p1 = a;
  p2 = p1;
  if (a > b)
        free (p1);
  p3 = (int *) malloc (sizeof (int));  // allocation 2
  *p3 = b;
  printf ("%d", *p2) ;
}
```

- what's wrong with this code ?
- what may happen at runtime ?

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**

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`

Use-after-free (example 1: information leakage)

```
char *login, *passwords;
login=(char *) malloc(...);
[...]
free(login); // login is now a dangling pointer
[...]
passwords=(char *) malloc(...);
   // may re-allocate memory area used by login
[...]
printf("%s\n", login)  // prints the passwords !
```

Use-after-free (example 2: execution hijacking)

```
typedef struct {
 void (*f)(void);   // pointer to a function
} st;

int main(int argc, char * argv[])
{
 st *p1;
 char *p2;
 p1=(st*)malloc(sizeof(st));
 free(p1); // p1 is now a dangling pointer
 p2=malloc(sizeof(int)); // memory area of p1 ?
 strcpy(p2,argv[1]);
 p1->f();  // calls any function you want ...
 return 0;
}
```

Use-after-Free, a typical heap vulnerability

CWE-416: `https://cwe.mitre.org/data/definitions/416.html`

Main characteristics:

- occurs when heap memory is explicitly allocated & de-allocated
  (garbage collection $\Rightarrow$ no dangling pointers)
- difficult to detect on the code: 3 distinct events (alloc, free and use)
  $\rightarrow$ need to check long execution paths
- exploitability depends on how predictable/controllable is the heap content
  (allocation strategy, heap spraying)

In practice:

- mostly targets web navigators (IE, Firefox, Chrome, etc.)
  - object langage programming
    objects $\Rightarrow$ # heap allocation + method tables in the heap
  - overlap of several heap memory allocators
    multi-language applications, custom allocators
- but other applications impacted as well !

See rule 8 of CERT C secure coding standard

Type confusion example [C++]

```
class Base {}; // Parent Class

class Exec: public Base {   // Child of Base Class
public: virtual void exec(const char *program)
         { system(program); }
};

class Print: public Base {   // Child of Base Class
public: virtual void sayHi(const char *str)
         { cout << str << endl; }
};

int main() {
    Base *b1 = new Print();
    Base *b2 = new Exec();
    Print *g;
          ...
    g = static_cast<Print*>(b1); // safe cast
    g->sayHi("hello world"); // call sayHi() function
          ...
    g = static_cast<Print*>(b2); // unsafe cast
    g->sayHi("/usr/bin/sh"); // call exec() function !
}
```

unsafe `Print` $\rightarrow_{upcast}$ `Base` $\rightarrow_{downcast}$ `Exec` conversion

Type confusion in practice

Yet another *type safey violation*:

intended type $\neq$ actual type

Occurs in some weakly typed compiled languages:

C: no checks when using `union` types

C++:

- *upcast* conversions always valid
- static verification of *downcast* conversion is NP-complete
  $\Rightarrow$ efficiency vs security trade-off is left to the user:
  - `reinterpret_cast`: no check
  - `static_cast`: only partial compile-time checks
  - `dynamic_cast`: complete run-time checks (performance penalty)

May occur as well is some interpreted languages (Java, JavaScript, ...) ...

... due to interpreter bugs !

Outline

Examples

Concatening command line arguments [C]

```
int main(int argc, char *argv[])
{ char name[2048];
strcpy(name, argv[1]);
strcat(name, " = ");
strcat(name, argv[2]); ... }
```

→ what may happen at execution ?

Listing the content of a directory [PHP]

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

→ how to remove the whole filesystem using this PHP script ?

```
; rm -rf /
```

A root cause to many exploits: improper input validation

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

- Buggy parsing & processing
  ex: invalid PDF file $\rightarrow$ buffer overflow $\rightarrow$ arbitrary code exececution

  input processing attack

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

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

  input injection attack

  Incorrect input $\Rightarrow$ forward an unsecure command to a back-end (database, OS, file system, Web browser, etc.)
  Untrusted facilities offered in many languages:
  C/C++ (`system`, `execv`, `ShellExecute`, etc.),
  Java (`Runtime.exec`), Perl, Python, JavaScript (`eval`), etc.

Why is it a problem ?

and possible solutions . . .

- numerous complex input formats
  file processing (PDF, Flash, jpeg, etc.), protocols, certificate (x.509)
  not always well-documentyed specification
  frequent updates and extensions . . .
  $\leadsto$ huge attack surface !
- parsers (too !) often written/updated/corrected by hand
  (without automated parser generator from well-defined formats)
- mix between parsing / (partial) validation / processing
  - sanitization may be spread along the code
    (beware of “time of check - time of use !)
  - no clear distinction between trusted/sanitized & untrusted data
- use of low-level input representations: strings
  $\rightarrow$ a single weakly typed reprsentation for many $\neq$ data
  (URLs, SQL commands, Unix commands, etc.)

  etc. . . .

As a (temporary) conclusion

Language level weaknesses exploitation

- no type safety:
  implicit type conversions, no conformance guarantee between "source types" and "runtime types"
- no memory safety: illegal memory accesses may occur at runtime
  → **spatial** vs **temporal** memory errors
- undefined behaviors, etc.

→ a long story: "Memory Errors: The Past, the Present, the Future" (V vd Veen at al)

⇒ **leads to unsecure binary code**

- binary encoding of integer and reals (overflows ? wrap-around ?)
- stack overflows (read/write/exec arbitrary data in the stack)
- heap vulnerabilities (read/write/exec arbitrary data in the heap)
- type confusion (read/write/exec arbitrary data in memory)
- and many others . . . !

Theses sources of unsecurity may be exploited by a (malicious) user,
with no extra knowledge than the code itself . . .

"simple" pgm crashes may often be turned on dangerous exploits !

Some interesting links

- Google Zero Project: 0day Exploit Root Cause Analyses
- From memory corruption to exploits [3]

[3] *SoK: External War in Memory* (L. Szekeres, M. Payer, T. Wei, D. Song) - 2013 IEEE S&P