UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Grenoble INP Ensimag

Software security, secure programming (and computer forensics)

Lecture 3: Programming languages (un)-security

Looking at the binary level

Master on Cybersecurity – Master MoSiG (HECS & AISSE)

Academic Year 2016 - 2017

Reminder

So far, we saw that:

- Unsecure softwares are (almost) everywhere . . .
- Programming languages (often) contribute to produce unsecure software:
  - misleading syntactic constructions
  - weak typing
  - undefined behaviors
  - etc.

But:

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

$\Rightarrow$ Let's have a look at the binary code level to answer . . .

Outline

The intruder

Arithmetic overflows

Stack-based vulnerabilities

Heap based vulnerabilities

Format string vulnerabilities

Inspecting the binary code

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 ; start a new application; etc.
- break application availability (Denial of Service)
  e.g., crash a server
- (silently) hide/inject a malware (Non Repudiation)
- etc.

Intruder model

How can operate an intruder when running an unsecure code ?

As a $\boxed{\text{regular user:}}$ **by (fully) controling the (regular) program inputs**

Examples:

- fully control the keyboard, the network, the input files content, etc.
- may control the environment variables, the file system, etc.
- cannot modify the code, break cryptography, etc.

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 $\leadsto$ `mov eax, ecx` or 686643 or `"DB+"` or . . .
- code + data lie in the same (physical) memory
  - but usually in distinct zones
  - usually the code zone cannot be over-written

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

numerous points where code and data meet together . . .
$\rightarrow$ 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 adress of an exception handler
- 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 ???

$\boxed{\text{wrap-around + undefined behavior + implicit conversions = a dangerous coktail!}}$

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 >= 0
}
```

$\rightarrow$ 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 . . .

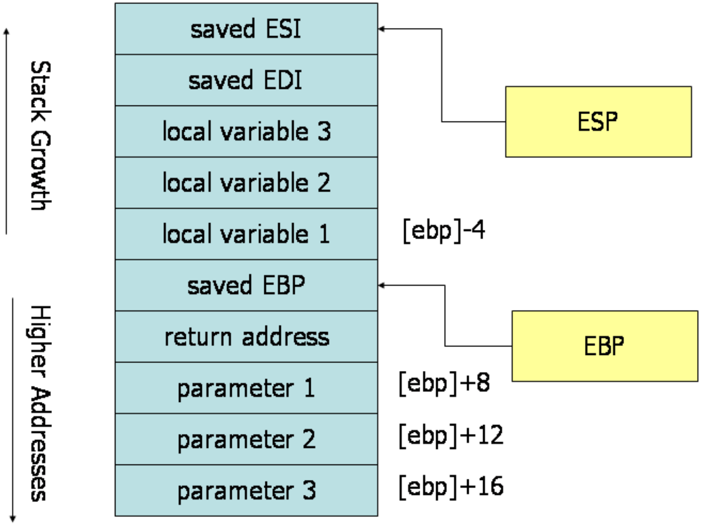
Memory organization at runtime

- 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<16;i++) t2[i]=2;
  strcpy(t1, t2) ;
  printf("La valeur de t : %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) ;
  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 . . .

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
- using uninitialized values accross several stack frames
- etc.

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)`
   $\hookrightarrow$ `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)
$\Rightarrow$ 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 f (int a, int b) 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 !
  (FTP server, graphic libraries, etc.)

Outline

A simple way to access “hidden” data …

Format functions in C

library functions: `printf`, `sprintf`, `fprintf`. etc.

- allow to convert data as (human readable) string representation
- functions with a **variable number of arguments**
  $\rightarrow$ can be called with an arbitrary number of parameters
- data to string conversion expressed by **string formats**, e.g,
  `"%x"` for hexadecimal values
  `"%d"` for decimal values
  `"%s"` for string, etc

At runtime:

- `printf ("hello %s", buf)`
  prints the content of `buf` as a string
- `printf ("hello %x", buf)`
  prints (the address) `buf` in hexa
- `printf ("hello %x")` prints ??
  … the hexa value of the “2nd parameter”
  $\rightarrow$ probably a value in the stack …

Example

```
void f (char src[])
{
int x = 1 ;
char buf [100];
snprintf (buf, sizeof(buf), src) ;
buf [sizeof(buf) -1] = '\0';
printf("%s \n", buf) ;
}

int main(int argc, char *argv[]) {
f (argv[1]) ;
}
```

- what's the result of `./a.out Bob)` ?
- what's the result of `./a.out "Bob %x %x")` ?

Possible consequences:

- information disclosure (print some memory content)
- denial of service (program crash if invalid memory access)

Outline

Understanding and analysing binary code ? (1/2)

```
01010100 01101000
01101001 01101110
01101011 00100000
01100100 01101001
01100110 01100110
01100101 01110010
01100101 01101110
01110100 00101110
```

```
00000000    push    ebp
00000001    mov     ebp, esp
00000003    movzx   ecx, [ebp+arg_0]
00000007    pop     ebp
00000008    movzx   dx, cl
0000000C    lea     eax, [edx+edx]
0000000F    add     eax, edx
00000011    shl     eax, 2
00000014    add     eax, edx
00000016    shr     eax, 8
00000019    sub     cl, al
0000001B    shr     cl, 1
0000001D    add     al, cl
0000001F    shr     al, 5
00000022    movzx   eax, al
00000025    retn
```

Disassembling !

Understanding and analysing binary code ? (2/2)

Recovering assembly-level code

- a non trivial task $\rightarrow$ static disassembling of x86 code undecidable (dynamic jumps, variable-length instructions, etc.)
- produce assembly-level IR instead of native assembly code $\rightarrow$ simpler language (a few instruction opcodes), explicit semantics (no side-effects), share analysis back-ends

Some existing tools

- IDA Pro
  a well-known commercial disassembler, # useful features
- On Linux plateforms (for ELF formats):
  - `objdump` (`-S` for code disassembling)
  - `readelf`
  - Debuggers can be used as well . . .
    ex: the `disass` command of `gdb`

x86 assembly language in one slide

Registers:

- stack pointer (ESP), frame pointer (EBP), program counter (EIP)
- general purpose: EAX, EBX, ECX, EDX, ESI, EDI
- flags

Instructions:

- data transfer (MOV), arithmetic (ADD, etc.)
- logic (AND, TEST, etc.)
- control transfer (JUMP, CALL, RET, etc)

Adressing modes:

- register: mov eax, ebx
- immediate: mov eax, 1
- direct memory: mov eax, [esp+12]

As a (temporary) conclusion

Language level weaknesses

- 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
- undefined behaviors, etc.

$\Rightarrow$ **lead to unsecure binary code**

- binary encoding of integer and reals (overflows ? wrap-around ?)
- stack overflows (read/write arbitrary data in the stack)
- heap vulnerabilities (read/write arbitrary data in the heap)
- format strings (read arbitrary data in the stack)
- 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 !