UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Grenoble INP Ensimag

Software security, secure programming (and computer forensics)

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

Master on Cybersecurity – Master MoSiG (HECS & AISSE)

Academic Year 2016 - 2017

Preamble

Bad news

Many 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 itself
- from the compiler / interpreter
- from the execution plateform

Outline

Programmer's level protections

Compilers level protections

Plateform level protections

Code hardening

Most language level vulnerabilities are known !

$\rightarrow$ there exist code patterns to mitigate their effects . . .

Examples

- 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)
- 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.

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 */
  }
  /* ... */
}
```

Example 2

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

Example of non compliant code

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

Example of compliant code

```c
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)
Tools: 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” . . .
Tools: Frama-C, Polyspace, CodeSonar, Fortify, etc.

Outline

Programmer's level protections

Compilers level protections

Plateform level protections

Compilers may help for code protection

Most compilers offer compilation options to help mitigating the effect of vulnerable code ...

$\rightarrow$ automatically generate extra code to enforce security

Examples

- stack protection
  - stack layout
  - canaries
  - shadow stack for return addresses
  - control-flow integrity
  - ...
- 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.

Outline

Programmer's level protections

Compilers level protections

Plateform level protections

Some more generic protections from the execution plateform

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

- Memory layout randomization (ASLR)
  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 ...
- 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 ...

Rk: exists other dedicated protections for more specific plateforms:
JavaCard, Android, embedded systems, TPMs, etc.

Conclusion

- $\exists$ numerous protections to avoid / mitigate vulnerability exploitations
- several protection levels
  code, verification tools, compilers, plateforms
- they allow to “compensate” 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 !

$\boxed{\rightarrow \text{ an endless game between “attackers” and “defenders” !}}$