Detecting and exploiting integer overflows

Guillaume TOURON

Laboratoire Verimag, Ensimag - Grenoble INP
Marie-Laure Potet, Laurent Mounier

20/05/11

Work subject

Subject

Binary code static analysis for vulnerabilities detection

- Focus on arithmetic problems

Application security is critical for information systems

- Programming bad practices

Goals

- Work with a professional environment : **IDA Pro**
- Develop some analysis to make easier vulnerabilities detection

Work subject

Subject

Binary code static analysis for vulnerabilities detection

- Focus on arithmetic problems

Application security is critical for information systems

- Programming bad practices

Goals

- Work with a professional environment : **IDA Pro**
- Develop some analysis to make easier vulnerabilities detection

Work subject

Subject

Binary code static analysis for vulnerabilities detection

- Focus on arithmetic problems

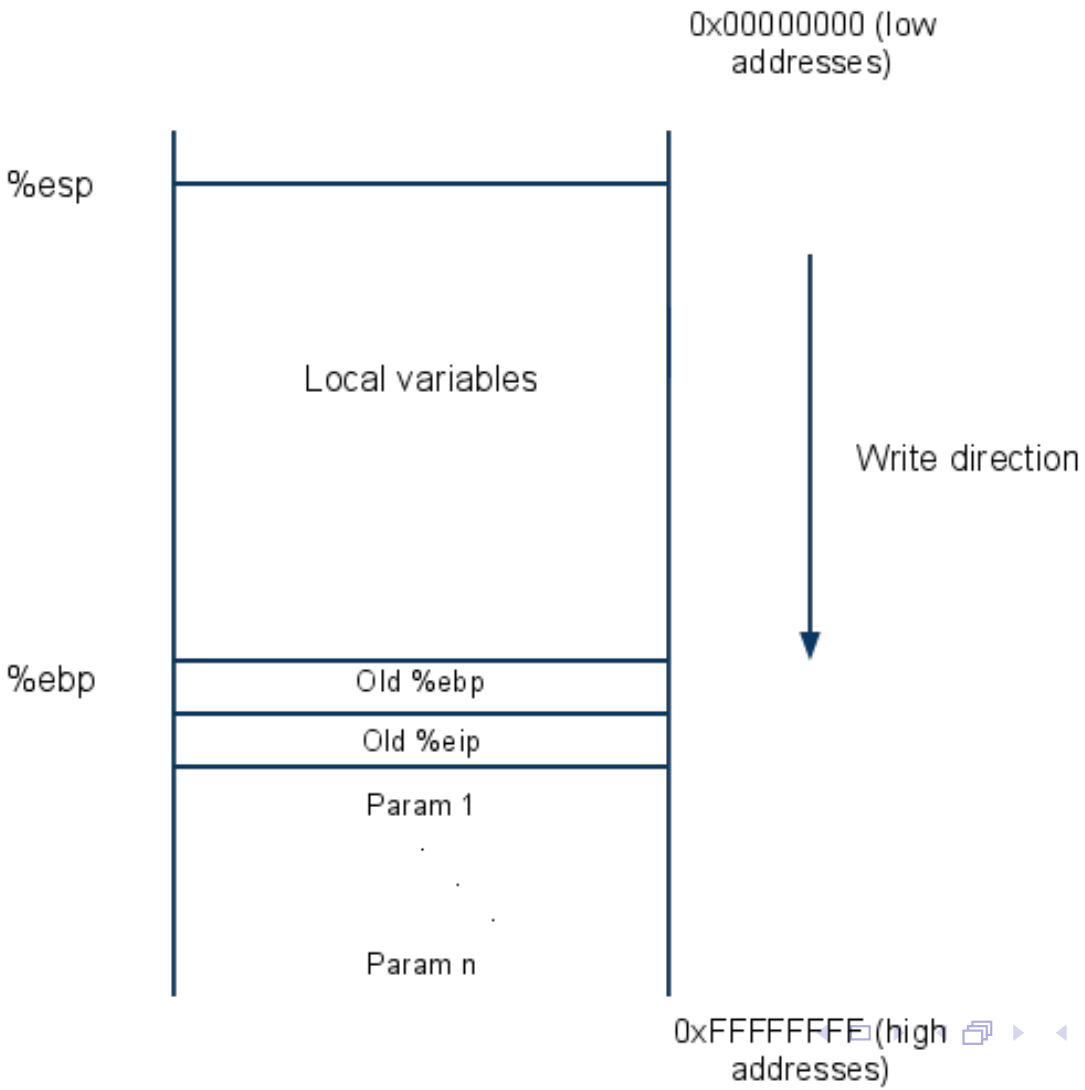
Application security is critical for information systems

- Programming bad practices

Goals

- Work with a professional environment : **IDA Pro**
- Develop some analysis to make easier vulnerabilities detection

Buffer overflow



Buffer overflow vulnerabilities

Exploitability

Integer overflow can lead to buffer overflow
Buffer overflow can lead to arbitrary code execution

Integer overflows and buffer overflows top ranked by CWE Exploitability (CWE):

- Buffer overflow: High to Very High (3^{rd})
- Integers overflow: Medium (16^{th})

Conclusion

We have to care about arithmetic overflow and avoid them

Buffer overflow vulnerabilities

Exploitability

Integer overflow can lead to buffer overflow
Buffer overflow can lead to arbitrary code execution

Integer overflows and buffer overflows top ranked by CWE Exploitability (CWE):

- Buffer overflow: High to Very High (3^{rd})
- Integers overflow: Medium (16^{th})

Conclusion

We have to care about arithmetic overflow and avoid them

Buffer overflow vulnerabilities

Exploitability

Integer overflow can lead to buffer overflow
Buffer overflow can lead to arbitrary code execution

Integer overflows and buffer overflows top ranked by CWE Exploitability (CWE):

- Buffer overflow: High to Very High (3^{rd})
- Integers overflow: Medium (16^{th})

Conclusion

We have to care about arithmetic overflow and avoid them

x86 integers binary representation

Basic C types on x86 32 bits:

	char	*short*	*int*	*long int*
signed	[-128,127]	[-32,768,32,767]	$[-2^{31}, 2^{31}-1]$	$[-2^{63}, 2^{63}-1]$
unsigned	[0,255]	[0,65535]	$[0, 2^{32}-1]$	$[0, 2^{64}-1]$

Signed values representation

For negative values, $MSB = 1$ (2's complement representation)

$$\text{e.g } -1 = 0xFFFFFFFF$$

x86 integers binary representation

Basic C types on x86 32 bits:

	char	*short*	*int*	*long int*
signed	[-128,127]	[-32,768,32,767]	$[-2^{31}, 2^{31}-1]$	$[-2^{63}, 2^{63}-1]$
unsigned	[0,255]	[0,65535]	$[0, 2^{32}-1]$	$[0, 2^{64}-1]$

Signed values representation

For negative values, $MSB = 1$ (2's complement representation)

$$\text{e.g } -1 = 0xFFFFFFFF$$

Dangerousness of misinterpreting

First issue

Small negative integers can be interpreted as huge integers

Dangerous cases:

- Sanity checks
- Copy operations
- Array indexations

Dangerous functions

Some famous functions: *strncpy*, *strncat*, *snprintf*, *memcpy*...
These functions take a length *unsigned* parameter

Dangerousness of misinterpreting

First issue

Small negative integers can be interpreted as huge integers

Dangerous cases:

- Sanity checks
- Copy operations
- Array indexations

Dangerous functions

Some famous functions: *strncpy*, *strncat*, *snprintf*, *memcpy*...
These functions take a length *unsigned* parameter

Dangerousness of misinterpreting

First issue

Small negative integers can be interpreted as huge integers

Dangerous cases:

- Sanity checks
- Copy operations
- Array indexations

Dangerous functions

Some famous functions: *strncpy*, *strncat*, *snprintf*, *memcpy*...
These functions take a length *unsigned* parameter

Dangerousness of misinterpreting

First issue

Small negative integers can be interpreted as huge integers

Dangerous cases:

- Sanity checks
- Copy operations
- Array indexations

Dangerous functions

Some famous functions: *strncpy*, *strncat*, *snprintf*, *memcpy*...
These functions take a length *unsigned* parameter

Dangerousness of misinterpreting

First issue

Small negative integers can be interpreted as huge integers

Dangerous cases:

- Sanity checks
- Copy operations
- Array indexations

Dangerous functions

Some famous functions: *strncpy*, *strncat*, *snprintf*, *memcpy*...
These functions take a length *unsigned* parameter

Dangerousness of misinterpreting

memcpy example

*void *memcpy(void *dest, const void *src, size_t n);*

⇒ What happens if this value is user-controlled?

Let's take an example

Bad

```
#define LEN 512
...
void vuln(char *src, int s) {
    char dst[LEN];
    int size = s;
    if(s < LEN) {
        memcpy(dst, src, size);
    }
}
...
vuln("Test", -1);
```

Dangerousness of misinterpreting

memcpy example

*void *memcpy(void *dest, const void *src, size_t n);*

⇒ What happens if this value is user-controlled?

Let's take an example

Bad

```
#define LEN 512
...
void vuln(char *src, int s) {
    char dst[LEN];
    int size = s;
    if(s < LEN) {
        memcpy(dst, src, size);
    }
}
...
vuln("Test", -1);
```

Dangerousness of misinterpreting

Analysis

We have $size = -1$ ($0xFFFFFFFF$)
CPU compares *size* and 512 as signed values
$\Rightarrow size < 512 == True$

Vulnerability

But *memcpy* takes a *unsigned* argument, so $size = 2^{32} - 1$
By consequences, a buffer overflow occurs

A potential attacker can take control of flow execution

Dangerousness of misinterpreting

Analysis

We have $size = -1$ ($0xFFFFFFFF$)
CPU compares *size* and 512 as signed values
$\Rightarrow size < 512 == True$

Vulnerability

But *memcpy* takes a *unsigned* argument, so $size = 2^{32} - 1$
By consequences, a buffer overflow occurs

A potential attacker can take control of flow execution

Dangerousness of misinterpreting

Analysis

We have $size = -1$ ($0xFFFFFFFF$)
CPU compares *size* and 512 as signed values
$\Rightarrow size < 512 == True$

Vulnerability

But *memcpy* takes a *unsigned* argument, so $size = 2^{32} - 1$
By consequences, a buffer overflow occurs

A potential attacker can take control of flow execution

Pattern matching

Patterns

We look for interesting (= dangerous) patterns

Some patterns:

- Calls to dangerous functions (*memcpy*, *strncpy*...)
  - Search signed comparisons on unsigned parameters
- Dangerous instructions

```
rep movsd
```

- Array indexation

```
movl $0x2a,-0x2c(%ebp,%eax,4)
```

Pattern matching

Patterns

We look for interesting (= dangerous) patterns

Some patterns:

- Calls to dangerous functions (*memcpy*, *strncpy*...)
  - Search signed comparisons on unsigned parameters
- Dangerous instructions

```
rep movsd
```

- Array indexation

```
movl $0x2a,-0x2c(%ebp,%eax,4)
```

Pattern matching

Patterns

We look for interesting (= dangerous) patterns

Some patterns:

- Calls to dangerous functions (*memcpy*, *strncpy*...)
  - Search signed comparisons on unsigned parameters
- Dangerous instructions

```
rep movsd
```

- Array indexation

```
movl $0x2a,-0x2c(%ebp,%eax,4)
```

Pattern matching

Patterns

We look for interesting (= dangerous) patterns

Some patterns:

- Calls to dangerous functions (*memcpy*, *strncpy*...)
  - Search signed comparisons on unsigned parameters
- Dangerous instructions

```
rep movsd
```

- Array indexation

```
movl $0x2a,-0x2c(%ebp,%eax,4)
```

Data dependencies

Looking for interesting data dependencies

- Sensitive parameters (e.g *size* from *memcpy*)
- Counter registers (e.g *%ecx* for *rep* prefixed instructions)

Analysis steps

- Scan code to find interesting data
  - Sensitive parameters (e.g *size* for *memcpy*)
- Backtrack these data for dependencies
- Apply code patterns to exhib vulnerabilities
  - Misinterpretation (e.g comparison as signed values)

Data dependencies

Looking for interesting data dependencies

- Sensitive parameters (e.g *size* from *memcpy*)
- Counter registers (e.g *%ecx* for *rep* prefixed instructions)

Analysis steps

- Scan code to find interesting data
  - Sensitive parameters (e.g *size* for *memcpy*)
- Backtrack these data for dependencies
- Apply code patterns to exhib vulnerabilities
  - Misinterpretation (e.g comparison as signed values)

Data dependencies

Looking for interesting data dependencies

- Sensitive parameters (e.g *size* from *memcpy*)
- Counter registers (e.g *%ecx* for *rep* prefixed instructions)

Analysis steps

- Scan code to find interesting data
  - Sensitive parameters (e.g *size* for *memcpy*)
- Backtrack these data for dependencies
- Apply code patterns to exhib vulnerabilities
  - Misinterpretation (e.g comparison as signed values)

Data dependencies

Looking for interesting data dependencies

- Sensitive parameters (e.g *size* from *memcpy*)
- Counter registers (e.g *%ecx* for *rep* prefixed instructions)

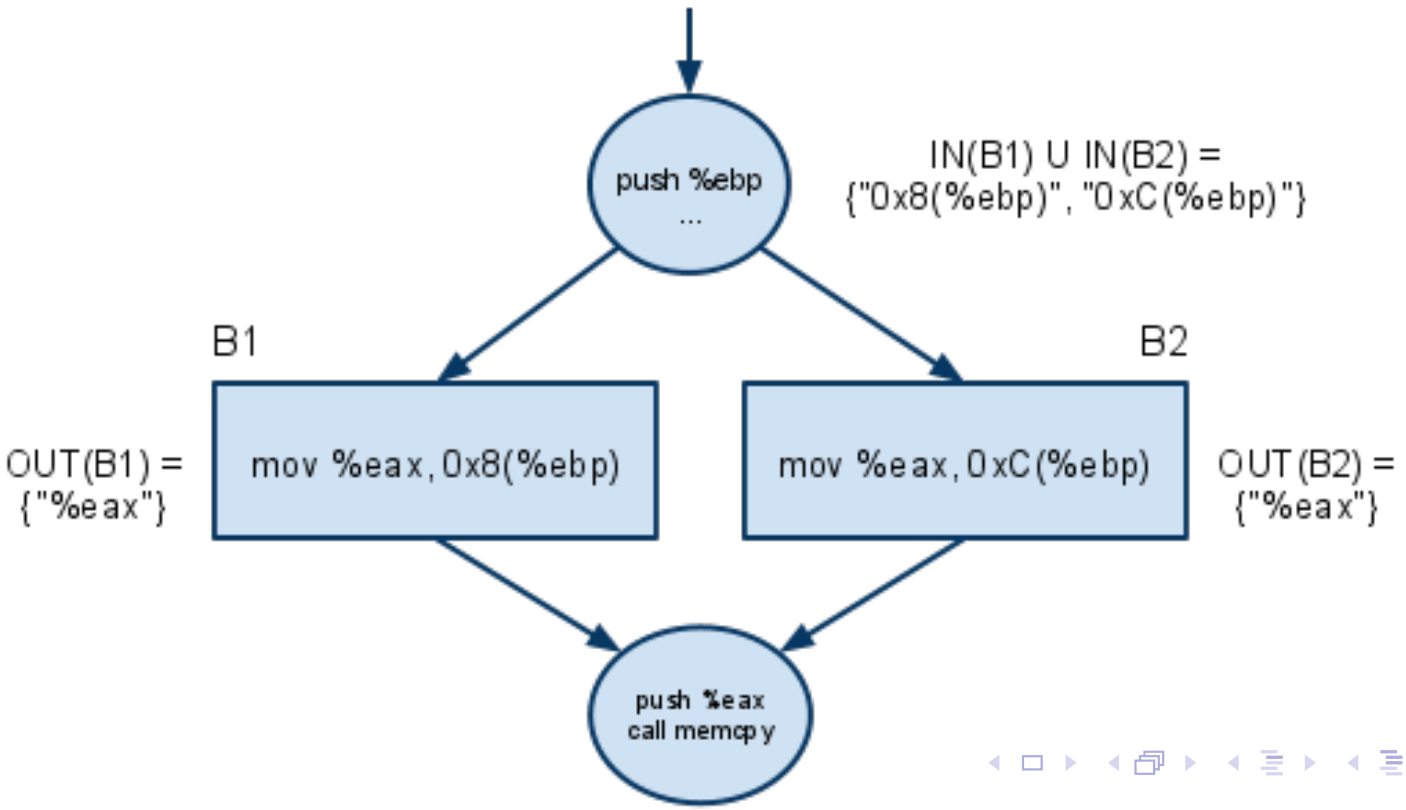
Analysis steps

- Scan code to find interesting data
  - Sensitive parameters (e.g *size* for *memcpy*)
- Backtrack these data for dependencies
- Apply code patterns to exhib vulnerabilities
  - Misinterpretation (e.g comparison as signed values)

Backward analysis

Dependencies

For a block B we have: $OUT(B) = \bigcup_{\forall S \in Successors(B)} IN(S)$



Backward analysis

Transfer function

Computes new tainted variables set for a basic block B:

$$IN(B) = F\_B(StmSeq, OUT(B))$$

We must define a subset of x86 (grammar)

$\Rightarrow$ Focus on instructions that imply dependencies

Examples:

- $mov[\epsilon|s|sx|zx]$
- Binary operations (*add*, *addc*, *sub*, *sbb*, *and*, *xor*, *or*...)

Backward analysis

Transfer function

Computes new tainted variables set for a basic block B:

$$IN(B) = F\_B(StmSeq, OUT(B))$$

We must define a subset of x86 (grammar)

$\Rightarrow$ Focus on instructions that imply dependencies

Examples:

- $mov[\epsilon|s|sx|zx]$
- Binary operations (*add*, *addc*, *sub*, *sbb*, *and*, *xor*, *or*...)

Backward analysis

Transfer function

Computes new tainted variables set for a basic block B:

$$IN(B) = F\_B(StmSeq, OUT(B))$$

We must define a subset of x86 (grammar)

$\Rightarrow$ Focus on instructions that imply dependencies

Examples:

- $mov[\epsilon|s|sx|zx]$
- Binary operations (*add*, *addc*, *sub*, *sbb*, *and*, *xor*, *or*...)

Environment

Several tools used:

- Binary analysis environment
  - IDA Pro
    Very used in security industry
    Powerful, many features available
    - CFG display
    - Several plugins
- API
  - First, IDAPython
    API for Python script in IDA Pro
  - Then, Paimei Framework
    Layer above IDAPython (easier to use)

Output example

Example on CVE-201-3970

```
[~] Search for predecessors: 0x5cb1fb46
[~] Previous bb: 0x5cb1fb21
push edi DEP: False
push eax DEP: False
call ds:__imp__CreateCompatibleDC@4 DEP: False
mov edi, eax DEP: True
cmp edi, ebx DEP: False
mov [ebp+var_4], edi DEP: True
jz loc_5CB1FCB8 DEP: False
[!] Pattern: 0x5cb1fbac : sbb eax, eax
[!] Pattern: 0x5cb1fbeb : cmp ecx, 100h : _CreateSizedDIBSECTION@28
```

Results

Pros:

- Automation
- Customization

Cons:

- False positive

Improvements:

- Improve data-flow analysis
  - Symbolic computation engine ?
- Add more dangerous code patterns
- Allow users to write their own patterns
  - Simple generic description language

Results

Pros:

- Automation
- Customization

Cons:

- False positive

Improvements:

- Improve data-flow analysis
  - Symbolic computation engine ?
- Add more dangerous code patterns
- Allow users to write their own patterns
  - Simple generic description language

Results

Pros:

- Automation
- Customization

Cons:

- False positive

Improvements:

- Improve data-flow analysis
  - Symbolic computation engine ?
- Add more dangerous code patterns
- Allow users to write their own patterns
  - Simple generic description language

General conclusion

Great subject, interesting people

First approach in research

- Documentation stage
  - Backward analysis
  - Vulnerabilities examples
- Implementation experimentation

Use new tools, techniques and frameworks

Q & A