Roberto Guanciale

Software Safety and Security DD2460

Memory Safety

Memory safety

- Prevent memory errors
- Very common problems for programs written using C/C++ (i.e. manual memory management and pointer arithmetic)
- Memory problems are one of the most dangerous security threat
  - XBox
  - Hearthbleed (OpenSSL)
  - Stagefright (Android media-stack)

Memory safety (in Java)

```
int a[] = new int[5];
int b;

a[20] = 10; // IndexOutOfBoundsException

a[1] == 0;  // true (array component are initialized with a default value)

a[1] = b;   // Compiler error: variable b might not have been initialized

List x = null;
x.add(a[1]);// NullPointerException

x = new Vector();
x.add(new Dog());

((Cat)x.get(0)).miaow(); // ClassCastException
```

Buffer Over-read/Overflow - basics

- Caused by programming error
  - Reads more data than the capacity
  - Allows more data to be stored than capacity
- Reading adjacent memory locations
  - leakage of secret data
- Overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker

Buffer Overread - Example

```c
void main(int argc, char ** argv) {
  char name[8];
  char pwd[8];
  int i,n = 0;

  strcpy(pwd, "pwd0");
  strcpy(name, argv[1]);
  n = atoi(argv[2]);

  printf("Echo ");
  for (i=0; i<n; i++) {
    printf("%c", name[i]);
  }
  printf("\n");
}
```

```
> ./main2 roberto 0
Echo
> ./main2 roberto 1
Echo r
> ./main2 roberto 2
Echo ro
> ./main2 roberto 7
Echo roberto
```

Buffer Overread - Example

```c
void main(int argc, char ** argv) {
  char name[8];
  char pwd[8];
  int i,n = 0;

  strcpy(pwd, "pwd0");
  strcpy(name, argv[1]);
  n = atoi(argv[2]);

  printf("Echo ");
  for (i=0; i<n; i++) {
    printf("%c", name[i]);
  }
  printf("\n");
}
```

```
> ./main2 roberto 0
Echo
> ./main2 roberto 1
Echo r
> ./main2 roberto 2
Echo ro
> ./main2 roberto 7
Echo roberto
> ./main2 roberto 16
Echo roberto�@
> ./main2 roberto 32
Echo roberto�@pwd0�TA��
```

Buffer Overread - Example

https://gist.github.com/simonwagner/10271224

Memory layout of a process

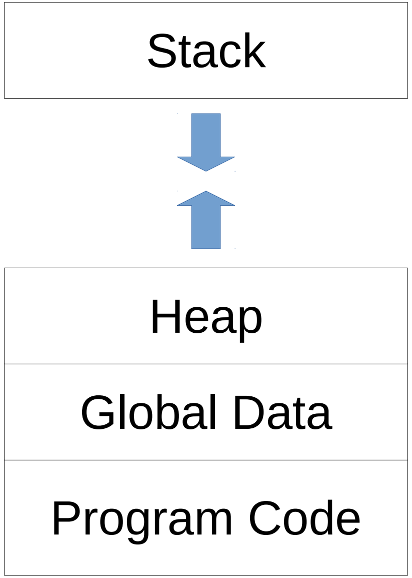
```c
int gloabl_var = 0;

int main(int argc, char ** argv) {
  int stack_var = 1;

  int * heap_var = malloc(sizeof(int));

  printf("Global: %p\n", &gloabl_var);
  printf("Stack:  %p\n", &stack_var);
  printf("Heap:   %p\n", heap_var);
  printf("Code:   %p\n", &main);
  printf("Lib:    %p\n", &malloc);
}
```

```
> ./main
Global: 0x601054
Stack:  0x7fffffffdcfc
Heap:   0x602010
Code:   0x4005f6
Lib:    0x4004f0
```



Stack

```
void hello(char * msg) {
  char buffer[16];
[redacted]
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

Stack

```
void hello(char * msg) {
  char buffer[16];
[redacted]
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto

Stack

```c
void hello(char * msg) {
  char buffer[16];
  printf("&msg adr \t%p\n", &msg);
  printf("msg adr \t%p\n", msg);
  printf("buffer adr \t%p\n\n", buffer);

  printf("enter the message for %s: \n", msg);
  printf("adr \t\t%p\n", *((void **)(buffer + 16)));
  printf("adr \t\t%p\n\n", *((void **)(buffer + 24)));

  gets(buffer);
  printf("message for %s is %s\n", msg, buffer);
  printf("adr \t\t%p\n", *((void **)(buffer + 16)));
  printf("adr \t\t%p\n\n", *((void **)(buffer + 24)));
  //*((unsigned int *)(buffer + 24)) = &hello;
  return;
}

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

0x7...d00: mainTag: Roberto

0x7...cd0: buffer: ???????

Stack

```
void hello(char * msg) {
  char buffer[16];
[redacted]
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto

0x7...cd0: buffer: ???????
0x7...cc8: msg: ??????

Stack

```
void hello(char * msg) {
  char buffer[16];
[redacted]
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto

0x7...cd0: buffer: ???????
0x7...cc8: msg: ??????

Stack

```
void hello(char * msg) {
  char buffer[16];
[redacted]
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto

0x7...cd0: buffer: ???????

0x7...cc8: msg: 0x7...d00

Stack

```c
void hello(char * msg) {
  char buffer[16];
[redacted]
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto

?

0x7...cd0: buffer: ???????

0x7...cc8: msg: 0x7...d00

Stack

```
void hello(char * msg) {
  char buffer[16];
[redacted]
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto

0x7...cd0: buffer: ???????

0x7...cc8: msg: 0x7...d00

Stack

```c
void hello(char * msg) {
  char buffer[16];
  printf("&msg adr \t%p\n", &msg);
  printf("msg adr \t%p\n", msg);
  printf("buffer adr \t%p\n\n", buffer);

  printf("enter the message for %s: \n", msg);
  printf("adr \t\t%p\n", *((void **)(buffer + 16)));
  printf("adr \t\t%p\n\n", *((void **)(buffer + 24)));

  gets(buffer);
  printf("message for %s is %s\n", msg, buffer);
  printf("adr \t\t%p\n", *((void **)(buffer + 16)));
  printf("adr \t\t%p\n\n", *((void **)(buffer + 24)));
  //*((unsigned int *)(buffer + 24)) = &hello;
  return;
}

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

0x7...d00: mainTag: Roberto

0x7...ce8: returnPtr: ????

0x7...cd0: buffer: ???????

0x7...cc8: msg: 0x7...d00

Stack

```c
void hello(char * msg) {
  char buffer[16];
[redacted]
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto

0x7...ce8: returnPtr: ????

0x7...cd0: buffer: ???????

0x7...cc8: msg: 0x7...d00

Stack

```
void hello(char * msg) {
  char buffer[16];
[redacted]
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto
0x7...ce8: returnPtr: 0x4006fb
0x7...ce0: framePtr: 0x7...d10
0x7...cd0: buffer: ???????
0x7...cc8: msg: 0x7...d00

Stack

```
void hello(char * msg) {
  char buffer[16];
[redacted]
  gets(buffer);
[redacted]
  return;
}

int main(int argc, char** argv) {
  char mainTag[16] = "Roberto";
[redacted]
  hello(mainTag);
}
```

0x7...d00: mainTag: Roberto
0x7...ce8: returnPtr: 0x4006fb
0x7...ce0: framePtr: 0x7...d10
0x7...cd0: buffer: ???????
0x7...cc8: msg: 0x7...d00

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

Global Data Overflow

- can attack buffer located in global data
- can manipulate critical data structures
- no return address
    - hence no easy transfer of control
- can target function pointers (e.g. C++ virtual tables, exception handlers)

Global Data Overflow

- can target function pointers (e.g. C++ virtual tables, exception handlers)

```
int compare_int( const void* a, const void* b) {
     int int_a = * ( (int*) a );
     int int_b = * ( (int*) b );
     if ( int_a == int_b ) return 0;
     else if ( int_a < int_b ) return -1;
     else return 1;
}
void do_damage() {...}
int (*my_compare)(const void*,const void*) = NULL;
char buffer[16];
int main() {
   my_compare = &compare_int;
   …
   qsort( a, 6, sizeof(int), my_compare);
}
```

Heap Overflow

- attack buffer located in heap
- no return address
- can target function pointers
- or manipulate critical data structures

Other memory safety problems

- Dangling pointers: do not point to a valid object of the appropriate type
  - wrong dynamic cast of pointers
  - missing update of pointers when memory is released (explicitly with free, implicitly by destroying the stack frame)
  - missing initialization of pointers
- Usage of non-initialized memory
- Memory leaks

Other memory safety problems

- Dangling pointers: do not point to a valid object of the appropriate type
  - wrong dynamic cast of pointers
  - missing update of pointers when memory is released (explicitly with free, implicitly by destroying the stack frame)
  - missing initialization of pointers
- Usage of non-initialized memory
- Memory leaks

```
Dog * x = malloc(sizeof(Dog));
add(list, x);
…
void * y = get(list,0);
miaow((Cat *) y);
```

Other memory safety problems

- Dangling pointers: do not point to a valid object of the appropriate type
  - wrong dynamic cast of pointers
  - missing update of pointers when memory is released (explicitly with free, implicitly by destroying the stack frame)
  - missing initialization of pointers
- Usage of non-initialized memory
- Memory leaks

```
Dog * x = malloc(sizeof(Dog));
free(x);
… // location of x is reused for allocating a Cat
woof(x);
```

Other memory safety problems

- Dangling pointers: do not point to a valid object of the appropriate type
  - wrong dynamic cast of pointers
  - missing update of pointers when memory is released (explicitly with free, implicitly by destroying the stack frame)
  - missing initialization of pointers
- Usage of non-initialized memory
- Memory leaks

```
void f() {
  int x = 42
  …
}
void g() {
  Dog * x; // Uninitialized
  woof(x);
}
void main() {
  f();g();
```

Other memory safety problems

- Dangling pointers: do not point to a valid object of the appropriate type
  - wrong dynamic cast of pointers
  - missing update of pointers when memory is released (explicitly with free, implicitly by destroying the stack frame)
  - missing initialization of pointers
- Usage of non-initialized memory
- Memory leaks

```
void f() {
  int pwd = 123456;
  …
}
int g() {
  int public_var; // Uninitialized
  return public_var;
}
void main() {
  f();printf(“%d”, g());
```

Other memory safety problems

- Dangling pointers: do not point to a valid object of the appropriate type
  - wrong dynamic cast of pointers
  - missing update of pointers when memory is released (explicitly with free, implicitly by destroying the stack frame)
  - missing initialization of pointers
- Usage of non-initialized memory
- Memory leaks

```
void f() {
   char * pwd = malloc(64);
   ...
   return
}
void main() {
   for (i=0; …; i++ ){
      f();
   }
}
```

Defenses (Prevention): Analysis tools (like valgrind)

- Instrument binary (almost every memory instruction)
  - keep track of
    - Validity (all unallocated memory starts as invalid)
    - Addressability (pointers to non-freed memory block)
  - replaces the standard C memory allocator (e.g. to delay reusage of memory)
- Can identify several problems (e.g. use of uninitialized memory, overflow on heap, accesses to freed memory, memory leaks)
- Need test input

Defenses (Detection): 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

Defenses (Detection): 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

Defenses (Detection): 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

Defenses (Detection): 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

Defenses (Detection):
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

Defenses (Prevention): 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

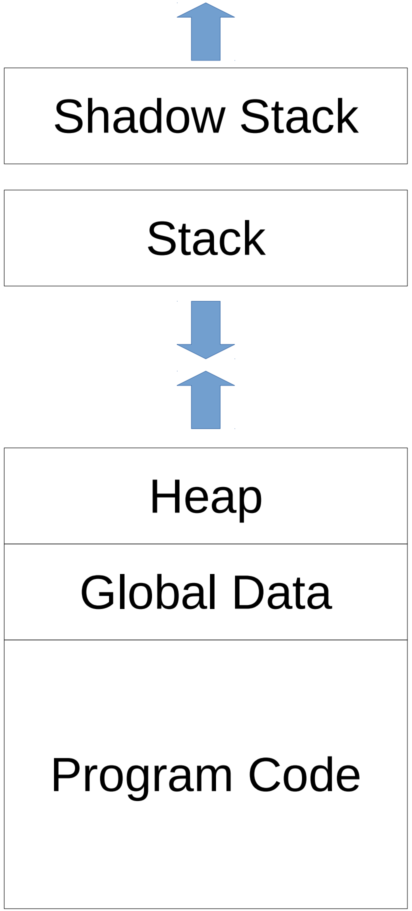
```
X = Y * Z;
```

```
X = Y*2;
X += Y*(Z-2);
```

```
int i;
X = 1;
for (i=0; i < Z; i++) {
  X = X + Y;
}
```

Defenses (Prevention): Stack protection

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



Defenses (Prevention): Executable Address Space Protection

- need HW support in MMU
- A page can be either writable or executable, but not both
  - Executable: program code
  - Writable: global data/stack/heap
- It is not possible to inject malware into executable code

Defenses (Prevention): Executable Address Space Protection

- need HW support in MMU
- A page can be either writable or executable, but not both
  - Executable: program code
  - Writable: global data/stack/heap
- It is not possible to inject malware into executable code
- issues: support for executable stack/heap code
  - needed for JIT (e.g. Java) or nested functions
  - need special provisions

Defenses (Prevention): 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

Defenses (Prevention): 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

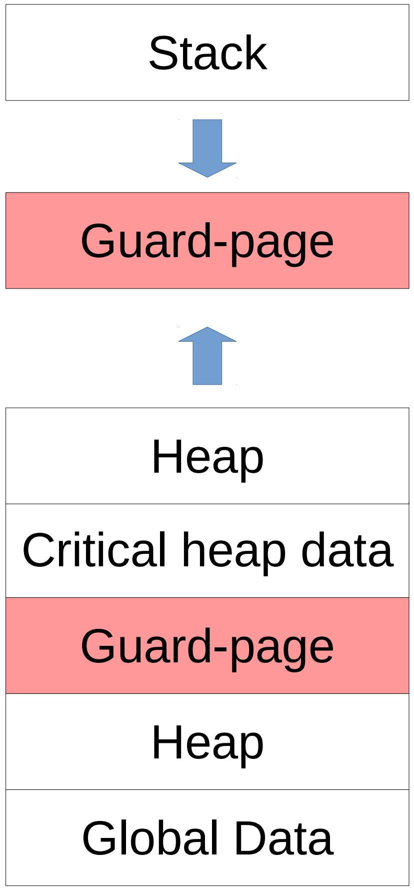
```
X = Y * Z;
```

```
X = Y*2;
X += Y*(Z-2);
```

```
int i;
X = 1;
for (i=0; i < Z; i++) {
  X = X + Y;
}
```

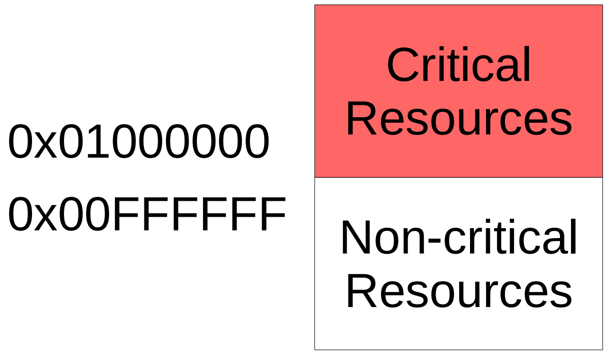
Defenses (Detection): Guard Pages

- 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
  - can be placed before critical data



Software Fault Isolation (Prevention)

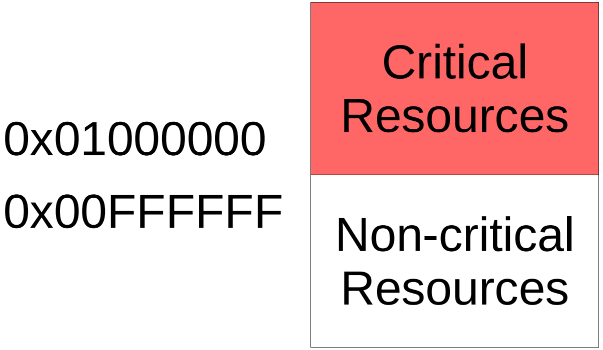
- Sandbox non-critical code
- Google Chrome Native Client
- Modify binary to ensure that overflows can not access critical resources



Software Fault Isolation (Prevention)

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

…
Store (X, Y)
...

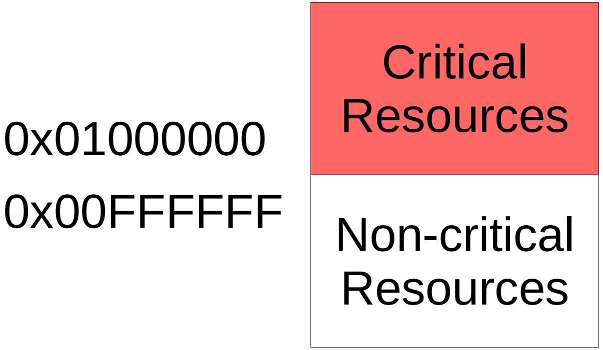


Software Fault Isolation (Prevention)

- 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 (Prevention)

- 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,Y)
...
```

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

Address	Memory
0x01000000	Critical Resources
0x00FFFFFF	Non-critical Resources

Software Fault Isolation (Prevention)

- 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,Y)
...
```

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

Address	Region
0x01000000	Critical Resources
0x00FFFFFF	Non-critical Resources

THANKS!

Any questions?

robertog@kth.se

Roberto Guanciale

Software Safety and Security DD2460

Other system security mechanisms

Other three security mechanisms

- Honeypots
- Malware-detection
- Port-scanning

Honeypots

- Detect and deflect attempts of unauthorized use of a systems
- Deploy fake data (or systems) that appears to be a legitimate
- Isolate data (e.g. never use it for legitimate functions)
- Monitor data
- Block accesses

Honeypots

- Detect and deflect attempts at unauthorized use of a systems
- Deploy fake data (or systems) that appears to be a legitimate
- Isolate data (e.g. never use it for legitimate functions)
- Monitor data
- Block accesses

For SPAM

- Register fake email addresses and use them in forums
- Never use these address for proper communications
- Monitor mail
- Report SMTP servers used for incoming e-mails

Honeypots

- Detect and deflect attempts at unauthorized use of a systems
- Deploy fake data (or systems) that appears to be a legitimate
- Isolate data (e.g. never use it for legitimate functions)
- Monitor data
- Block accesses

For SQL-Injection

- Create fake tables/records (e.g. fake users)
- Never use these tables for proper functions
- Monitor accesses to data
- Block request accessing them

Honeypots for memory safety

- Stack Canaries (canaries are not read or modified by licit code)
- Define pointers to unmapped virtual-addresses
  - If attacker discovers pointer, accesses will raise data-abort
- Deploy unused functions mapped as non-executable
  - If attacker discovers function, execution will raise exception
- Deploy application in a safe environment (e.g. virtualized environment), allowing attacker to compromise it and trigger alerts (e.g. by accessing monitored files)

Malware detection (execution of illicit code)

- Monitor behavior of application
  - Possibly in conjunction of honeypots
- Code signature
  - When an application is executed, check signature
    - Using a database of valid signatures
    - Using public key of application's vendor
  - Problem with JIT

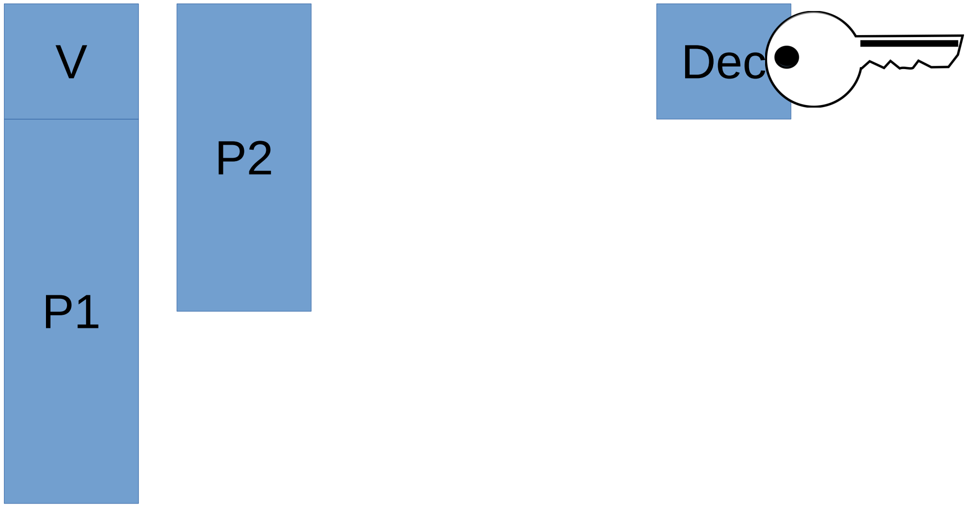
Malware detection (execution of illicit code)

- Malware Code signature
  - When an application is executed, check signature using a database of known viruses

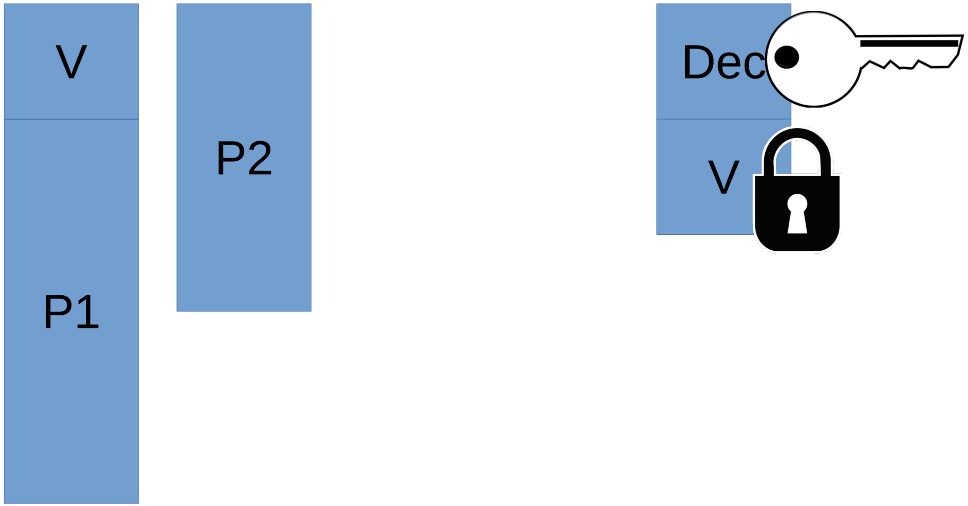
Malware detection
(execution of illicit code)

- Malware Code signature
  - When an application is executed, check signature using a database of known viruses
- Malware can encrypt itself
- At infection time
  - Generate key
  - Encrypt the malware body
  - Modify the bootstrap
  - Copy the bootstrap (decryption engine with key) and
  - the encrypted virus body
- At execution time
  - Decrypt the virus body
  - Execute payload

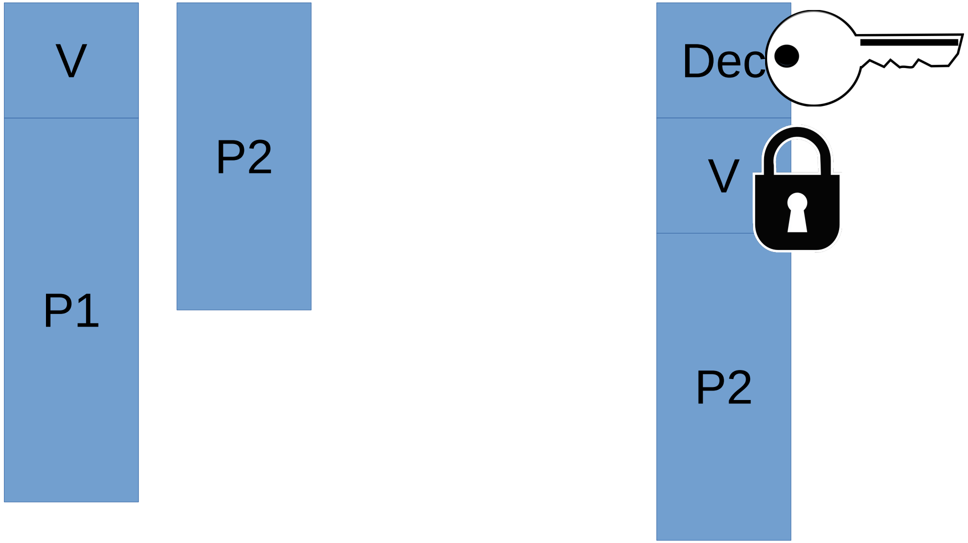
Malware detection
(execution of illicit code)



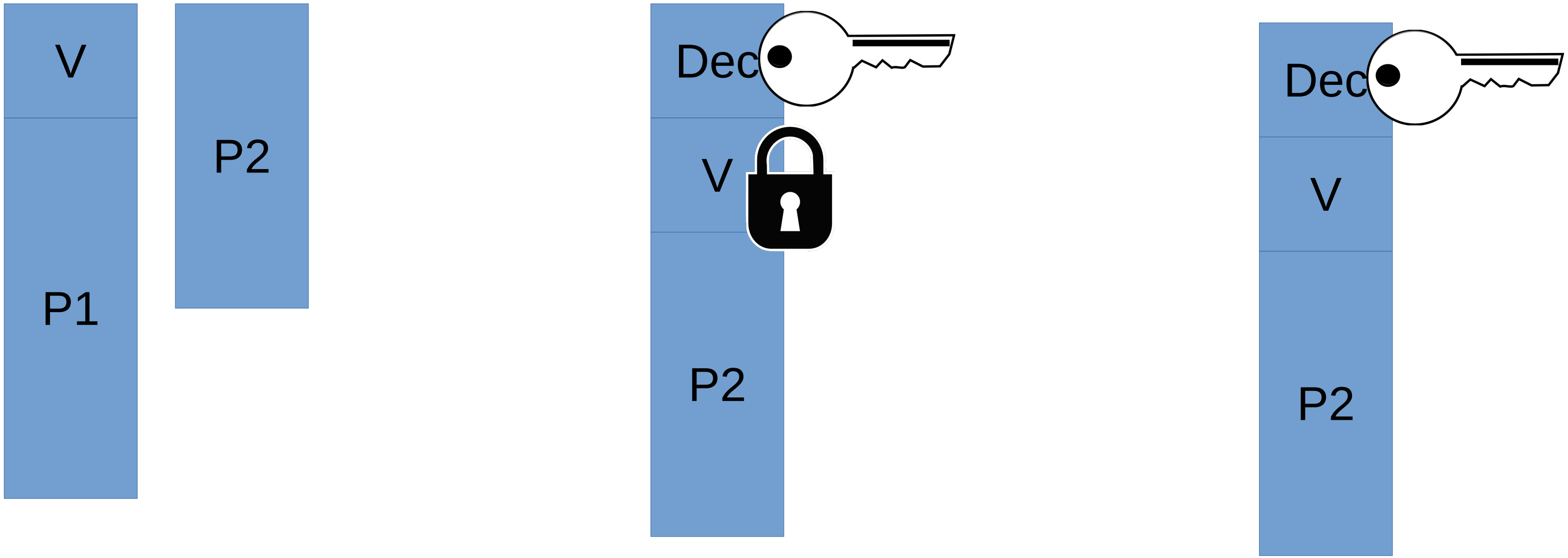
Malware detection
(execution of illicit code)



Malware detection (execution of illicit code)



Malware detection (execution of illicit code)



Malware detection (execution of illicit code)

- Malware Code signature
  - When an application is executed, check signature using a database of known malwares
- Malware can encrypt itself
- Generic descriptors
  - Use W ^ X policy
  - When a page is made executable it is not writable
    - Malware has decrypted itself
    - Scan the page for known malwares

Portscanning

- Today, a common attacker goal is to obtain remote access
- A port scanner is an application designed to probe a host for IP open ports
    - A port scan sends client requests to a range of server port, with the goal of finding an active port
- SYN-scans (to find listening TCP servers)
- UDP scanning (difficult due to connectionless)
- FIN-scans (received RST messages disclose closed ports)

Portscanning

- The good:
  - Can be used to remotely monitor an host
  - Identify if an attacker get control an opened ports
  - Discover services running in a IoT device
- The bad:
  - Commonly used by attackers to find victims
    1) Find a vulnerability in a service (e.g. a buffer overflow or missing input validation in handling network packets)
    2) Scan the network for hosts running the vulnerable service
    3) Attack the service

THANKS!

Any questions?

robertog@kth.se