University of BRISTOL

CyBOK

Cyber Security Body of Knowledge: Software Security

Frank Piessens

KU Leuven

bristol.ac.uk

University of BRISTOL

CyBOK

The CyBOK project would like to understand how the CyBOK is being used and its uptake. The project would like organisations using, or intending to use, CyBOK for the purposes of education, training, course development, professional development etc. to contact it at contact@cybok.org to let the project know how they are using CyBOK.

bristol.ac.uk

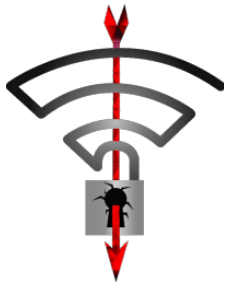
Software security

CyBOK

- The field of software security studies the problem of:
  - Maintaining **good properties** of software systems in the presence of **intelligent adversaries** trying to break these properties
- Rigorous study of a security problem requires three ingredients:
  - A **system model**:
    - a rigorous description of the system we are trying to secure.
  - A **security objective**:
    - what good properties of the system do we want to maintain under malicious attacker behavior?
    - Or what bad things do we want to prevent?
  - An **attacker model**:
    - a precise definition of the power of the attacker.

Software vulnerabilities

- In practice, software systems often do not have an explicit security objective
- Instead, software security is often about (avoiding) specific bugs that can lead to significant disruption



- The software security KA is about such “implementation vulnerabilities”:
  - Important categories of vulnerabilities
  - And how to prevent, detect or mitigate them

Overview of this seminar

CyBOK

This seminar mainly focuses on explaining three categories of vulnerabilities:

- Memory management vulnerabilities
- Structured output generation vulnerabilities
- Side-channel vulnerabilities

The Software Security KA document in addition discusses

- Several other categories of vulnerabilities
- Countermeasures
  - Prevention
  - Detection
  - Mitigation

Classifying implementation vulnerabilities

CyBOK

- Implementation vulnerability =
  - A defect in software code (a bug) that enables a specific attack technique
- Around 100.000 such vulnerabilities listed in the Common Vulnerabilities and Exposures (CVE) list:
  - Buffer overflows, SQL injection, cross-site scripting, race conditions, side-channel vulnerabilities, information leaks, incomplete access mediation, cross-site scripting, double free, . . .
- We discuss three classes in detail
  - the Cybok Software Security KA discusses more

1. Memory management vulnerabilities

CyBOK

- C-like languages offer mutable state that can be allocated, deallocated and accessed in a number of ways:
  - Local and global variables, malloc() / free(), …
  - Access through pointers and array indexing
- These memory management and access operations should be used correctly, e.g.:
  - Access arrays within bounds
  - Do not access memory after it has been deallocated
- For performance, compilers do not detect invalid memory accesses
  - Instead, behavior of the program becomes **undefined**
  - A program that can perform such an invalid access has a **memory management vulnerability**

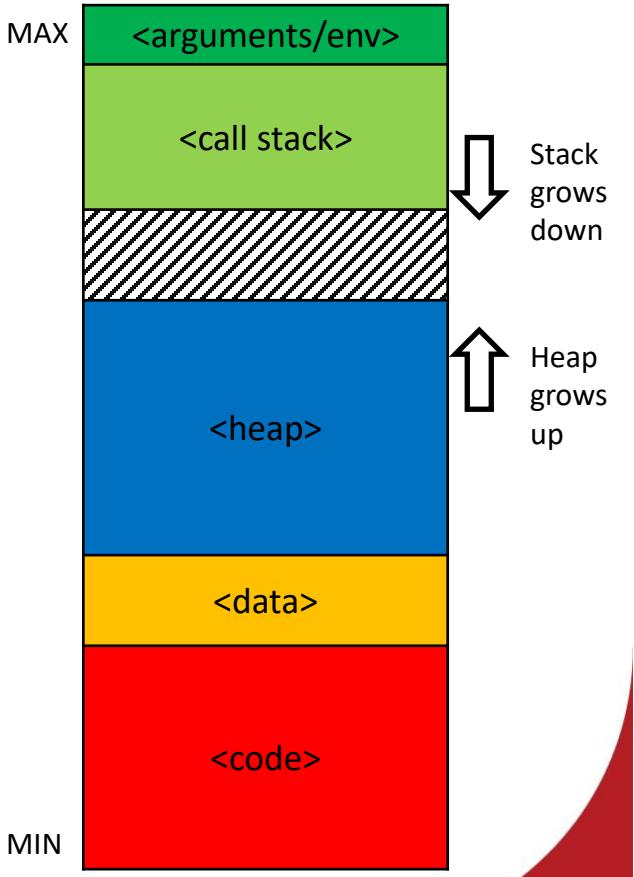
System model

CyBOK

- The system under attack is a C program that has been compiled to run on a modern processor
- Hence, the details of the system under attack vary with
  - The underlying platform (processor, operating system, …)
  - The compiler used
- But fortunately the general structure of processors, operating systems and compilers is sufficiently similar

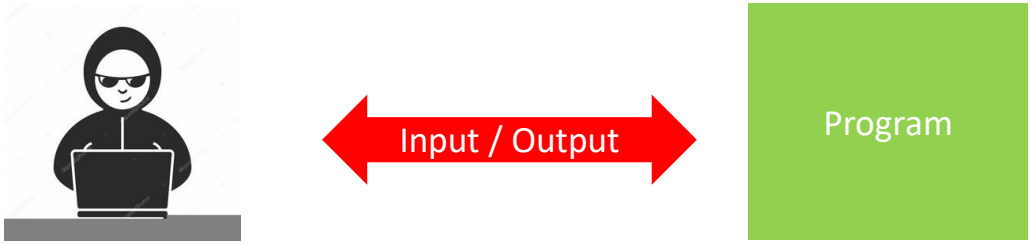
Compilation

- Each function is compiled separately and the resulting machine code is stored in a **code** section in memory
- Control-flow through the program is tracked by means of a **call stack**
- Variables used in the program are allocated in a number of ways:
  - Local variables are allocated on the call stack
  - Global variables are allocated in a dedicated **data** section in memory
  - Dynamic allocation is handled by a memory management library that manages a **heap**
    - malloc(), free(), …
- Pointers are represented as integer addresses, supporting pointer arithmetic
- Arrays are represented as pointers, indexing is similar to pointer arithmetic



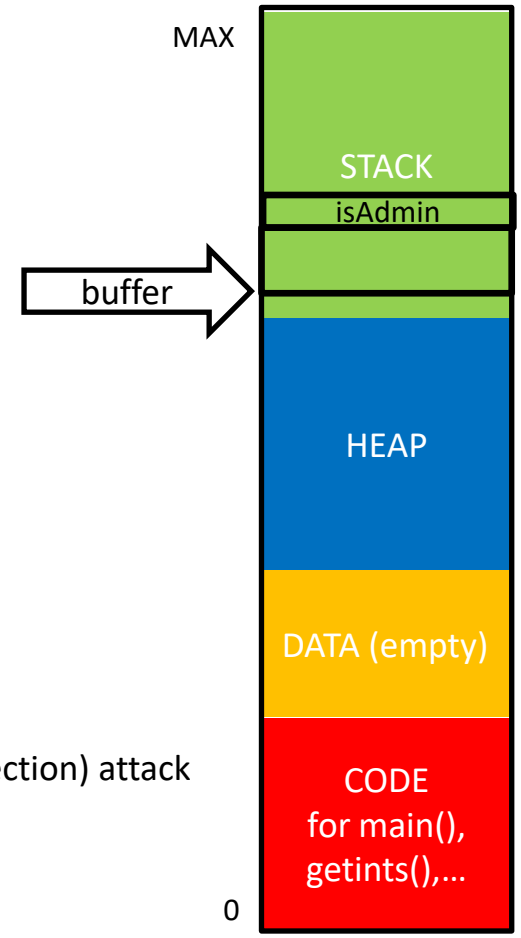
Attacker model

- We consider attacks that consist of crafting malicious input and learning from output of the program
  - Attacker knows the code and system software stack of the victim
  - Attacker can send arbitrary input and inspect resulting output



Memory corruption attacks

```
#include <stdio.h>
int main() {
  int isAdmin = 0;
  char buffer[10];
  gets(buffer);
  if (isAdmin != 0)
    printf("you are administrator!\n");
}
```



Many variants exist:

- Data-only attack
- Code corruption attack
- Direct code injection attack
- Code reuse (indirect code injection) attack

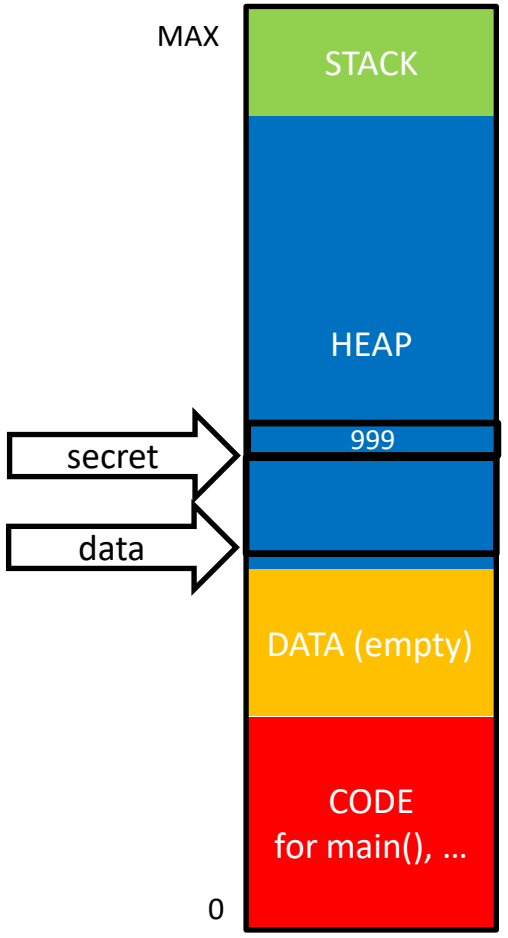
Memory disclosure attacks

```
#include <stdio.h>
#include <stdlib.h>

void main() {
  int i;
  int* secret = malloc(sizeof(int)); *secret = 999;
  // ...
  int* data = malloc(10 * sizeof(int));

  data[0] = 1; data[1] = 13; // ... store public data

  scanf("%d",&i);
  printf("%d\n", data[i]);
}
```



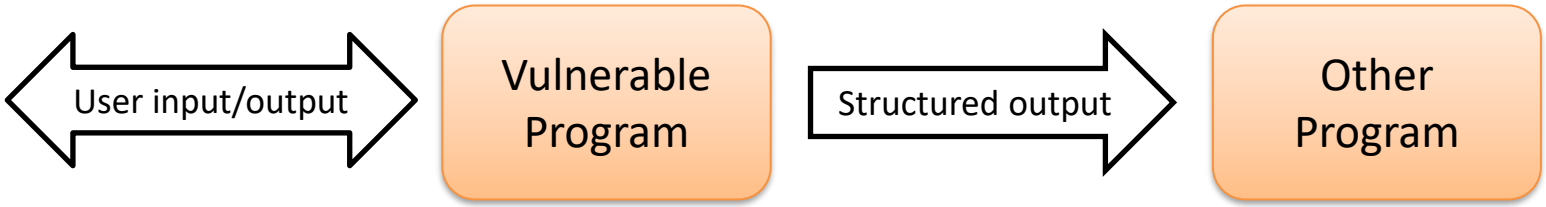
2. Structured output generation vulnerabilities

CyBOK

- (aka: injection vulnerabilities)
- Programs often construct structured output (e.g. SQL) using string concatenation
- When some of the strings can be chosen by an attacker, maliciously chosen values can change the structure of the output in unintended ways
- Examples: SQL injection, script injection (XSS), command injection, …

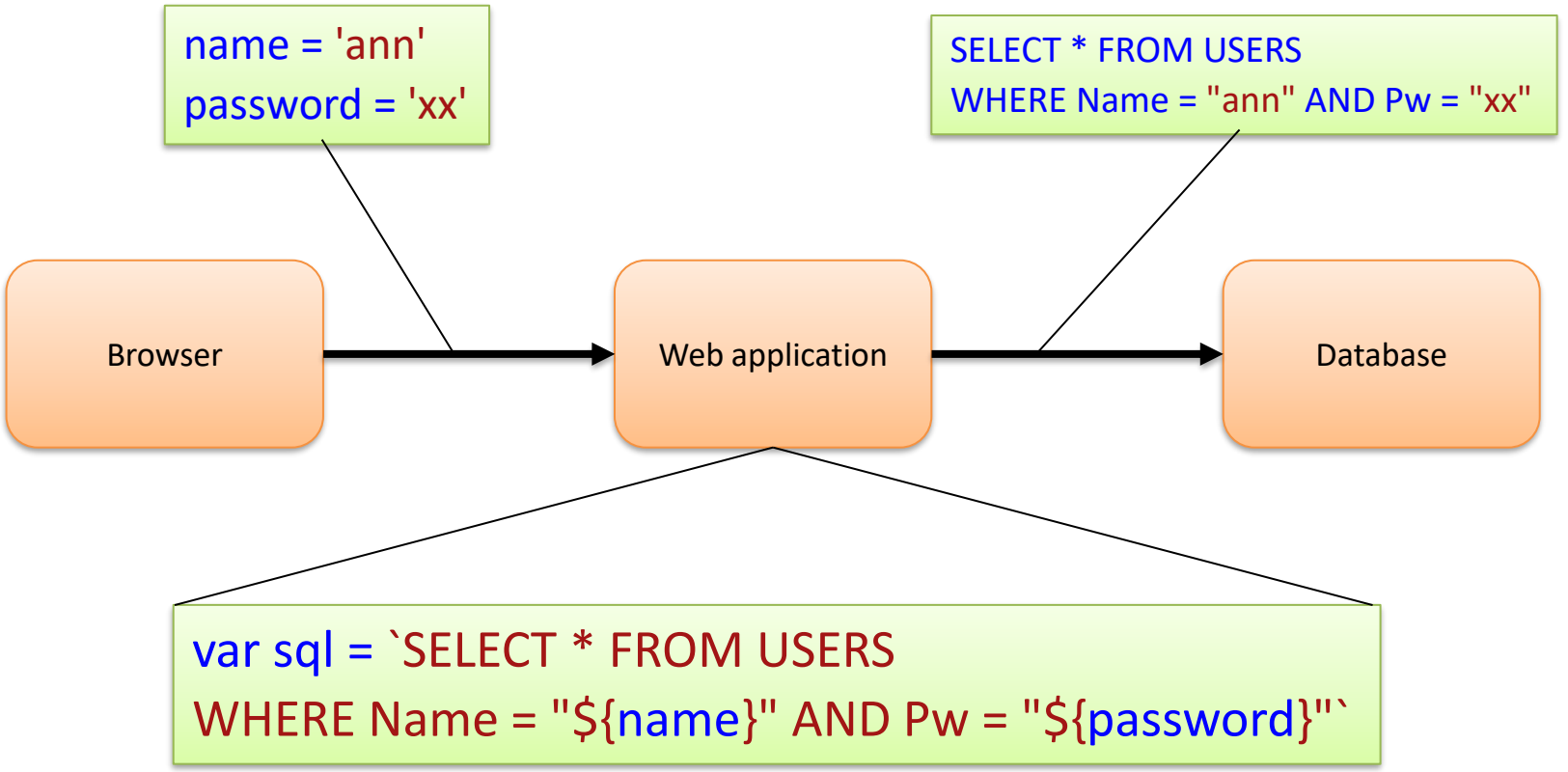
System model and attacker model

- System model

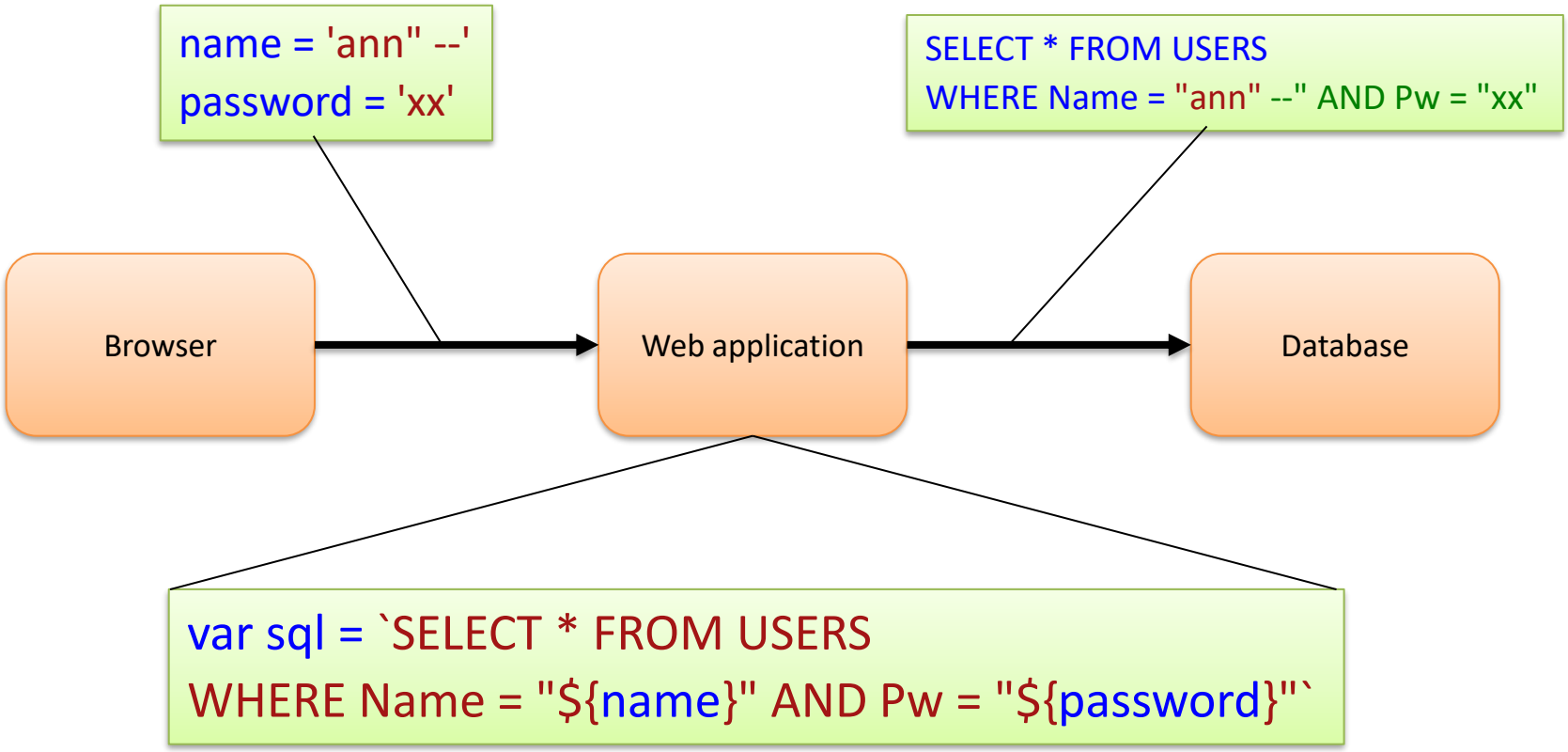


- Attacker model
  - Same attacker model as for the previous class:
    - Attacker knows the code of the system
    - Attacker can provide user input and read user output

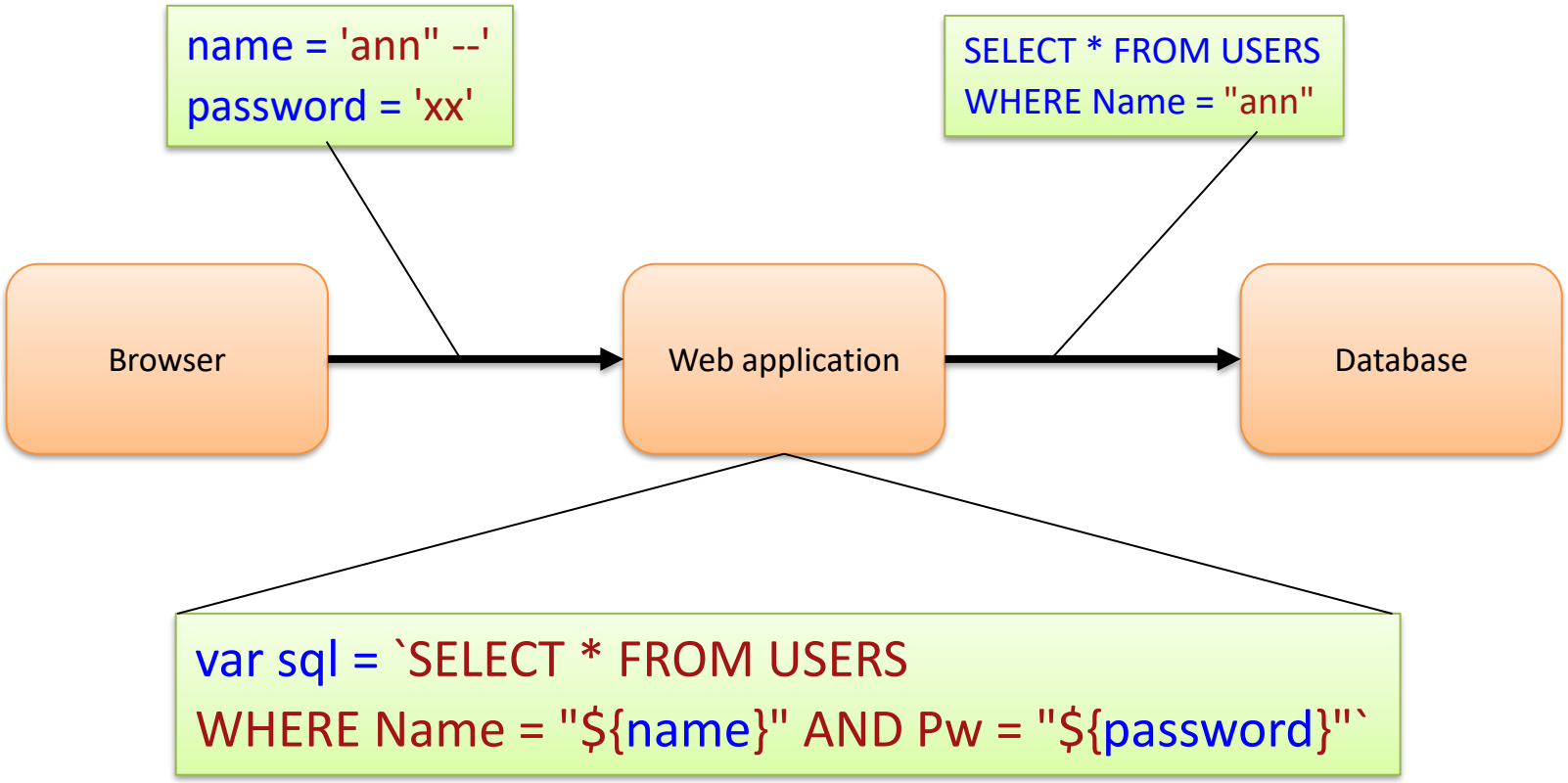
SQL injection attack



SQL injection attack



SQL injection attack



Additional complications

CyBOK

- Structured output generation vulnerabilities may seem conceptually simple, but several factors can contribute to the difficulty of avoiding them:
  - Structured output in languages that have sublanguages with a different structure, e.g. HTML with JavaScript, CSS, SVG, …
  - Structured output generation can happen in different phases, where output of one phase is later used as input for the next phase
    - Stored injection vulnerabilities
    - Higher-order injection vulnerabilities

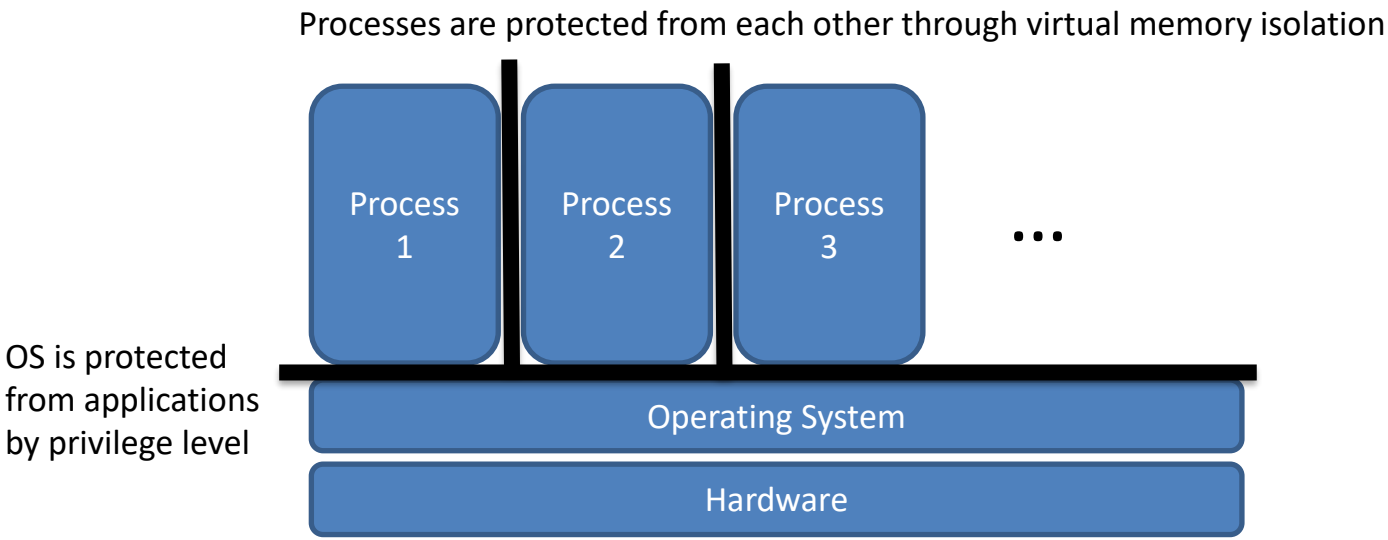
3. Side-channel vulnerabilities

CyBOK

- A **side-channel** is an information channel leaking information about the execution of a program through effects not intended for communication
  - E.g. timing, power consumption, electro-magnetic radiation, …
- Many side-channels require physical access to the system running the program under attack.
- **Software-based** side-channels can be read by software running on the same system as the program under attack
- A very recent important class of such channels exploit effects of the processor implementation.
  - Software-based micro-architectural side-channels

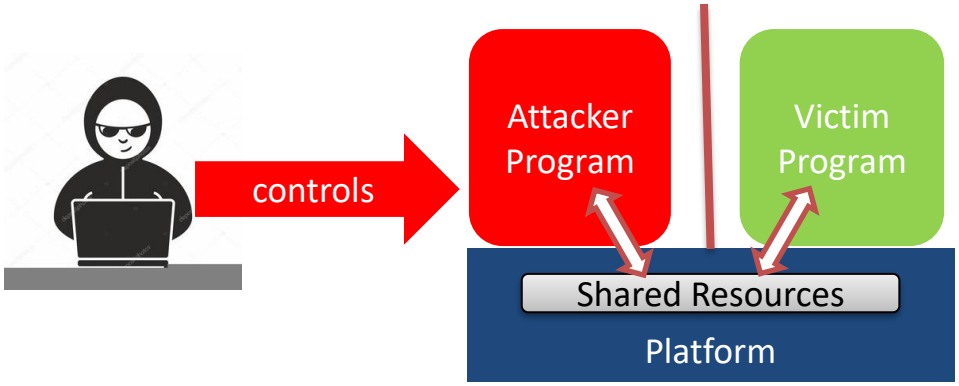
System model: a shared platform

- A platform runs programs from multiple stakeholders
  - Isolation mechanism isolates these programs
  - The platform supports communication between these programs



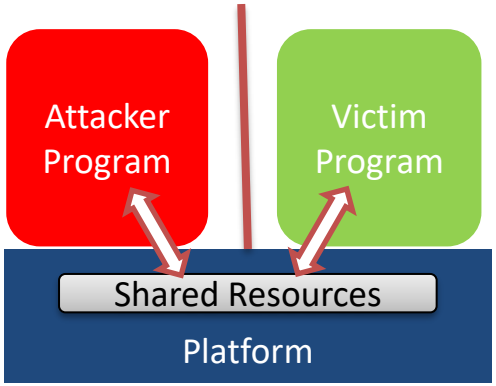
Attacker model

- The attacker can run code on the same platform where victim code is running.
- The objective of the attacker is to learn more about the victim than what one can learn through intended communication interfaces.

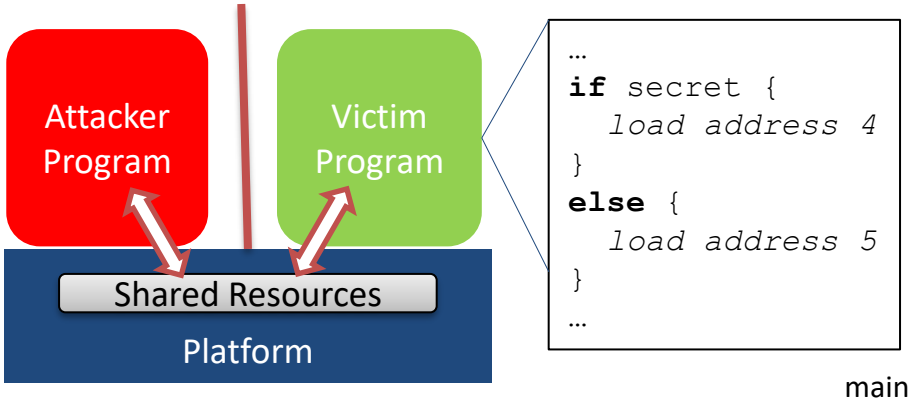


Micro-architectural attacks

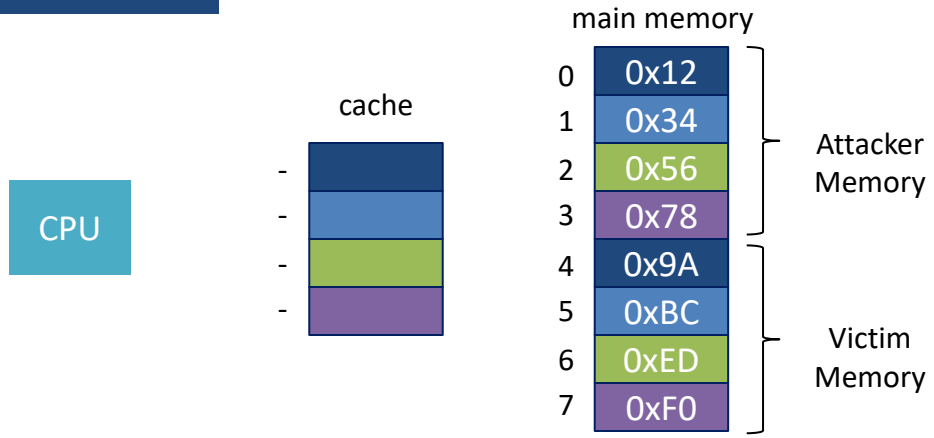
- The attacker uses shared *micro-architectural* resources
  - Architectural state: state as observed by software (memory, registers, …)
  - Micro-architectural state: additional state in the processor implementation, usually for performance (caches, branch predictors, …)



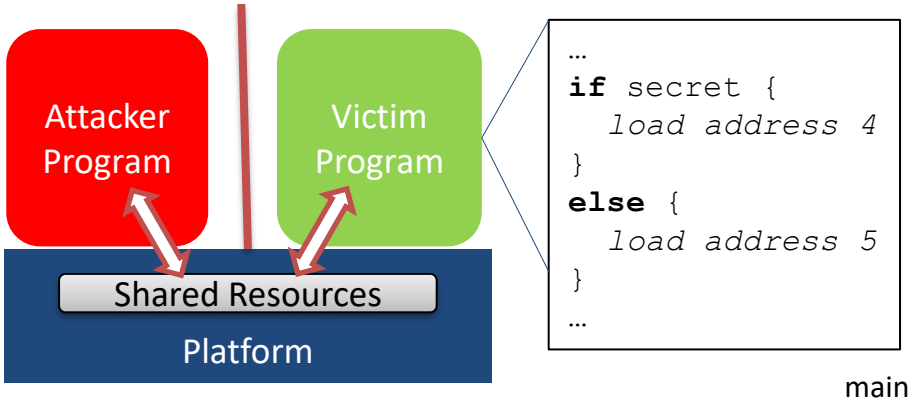
Side-channels: a simple example of a cache-attack



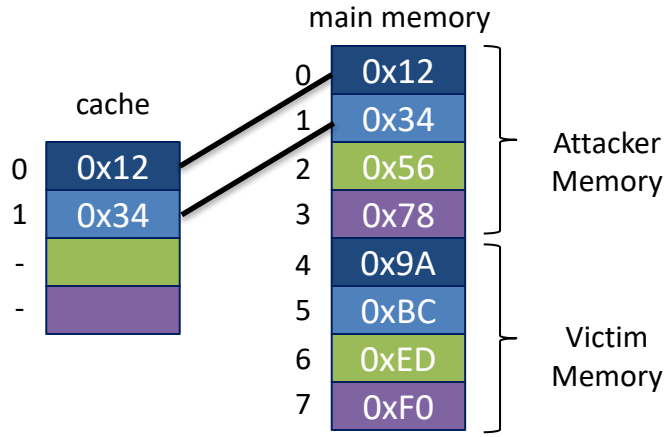
- The shared resources between attacker and victim program include a direct-mapped cache



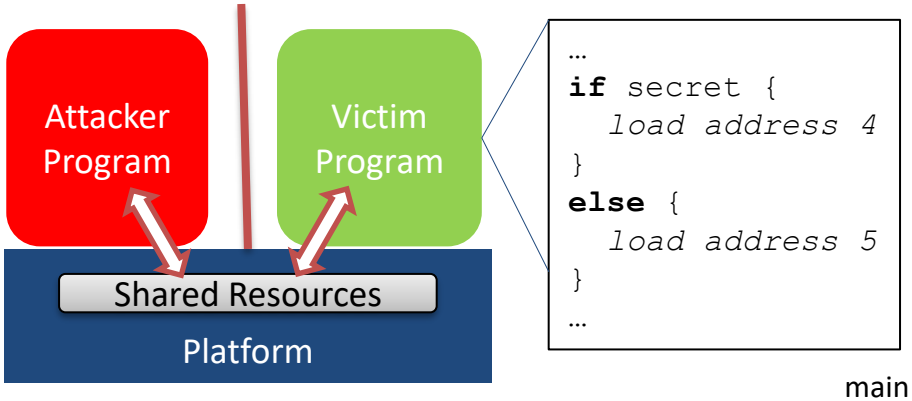
Side-channels: a simple example of a cache-attack



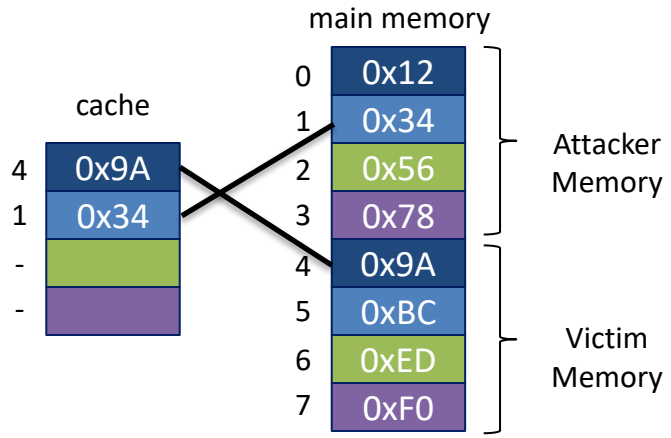
- The shared resources between attacker and victim program include a direct-mapped cache
  - First the attacker program runs and occupies the first two cache lines



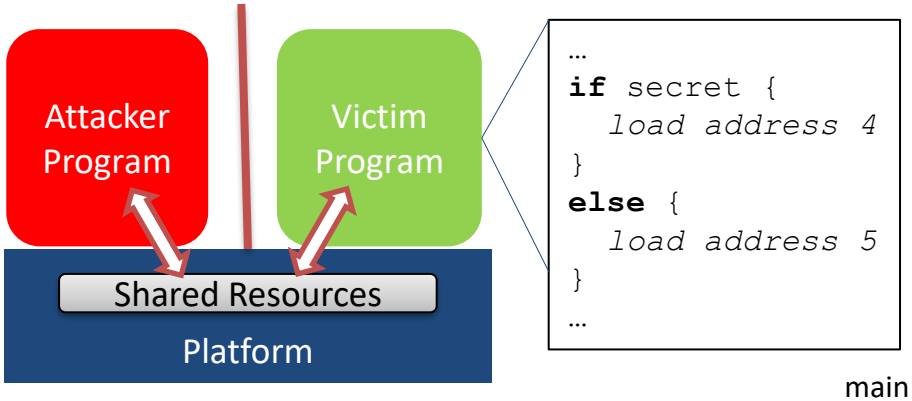
Side-channels: a simple example of a cache-attack



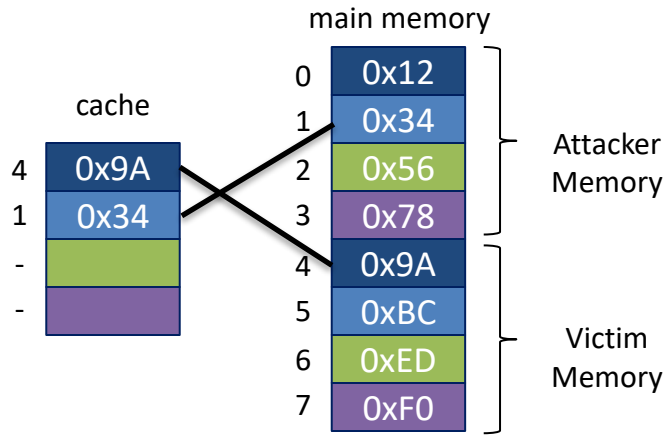
- The shared resources between attacker and victim program include a direct-mapped cache
  - First the attacker program runs and occupies the first two cache lines
  - Next the victim program runs and performs **secret-dependent** memory accesses



Side-channels: a simple example of a cache-attack



- The shared resources between attacker and victim program include a direct-mapped cache
  - First the attacker program runs and occupies the first two cache lines
  - Next the victim program runs and performs **secret-dependent** memory accesses
  - Finally, attacker measures duration of an access to address 0



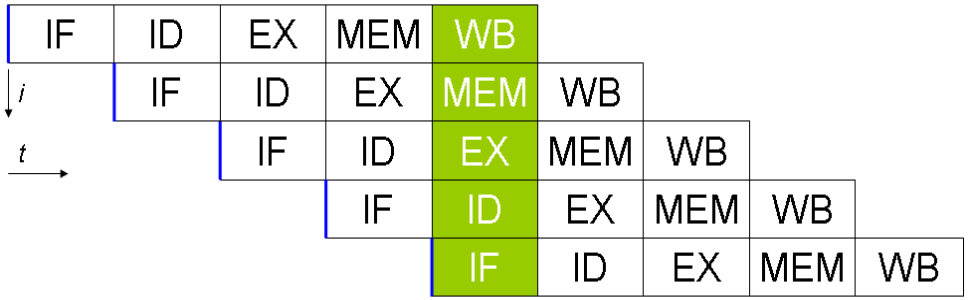
Transient execution attacks

CyBOK

- Transient execution attacks amplify the impact of existing side-channels **by giving the attacker control over the sending side of the channel** too
- The key observations are:
  - Processors are pipelined and sometimes execute instructions *speculatively*
    - No architectural effects are visible until instruction is committed
    - **Transient** execution is any execution that never gets committed
  - Transiently executed instructions *also impact the micro-architectural state*
  - The attacker *can influence what instructions get executed transiently*

Speculative execution

- All major processors support speculative execution
  - Processor implementations are pipelined
  - To keep the hardware busy, instructions are executed *out-of-order* and *speculatively*
  - If speculation was wrong, it gets rolled back
  - Such *transient* execution has no visible *architectural* effects, but there are persistent micro-architectural effects



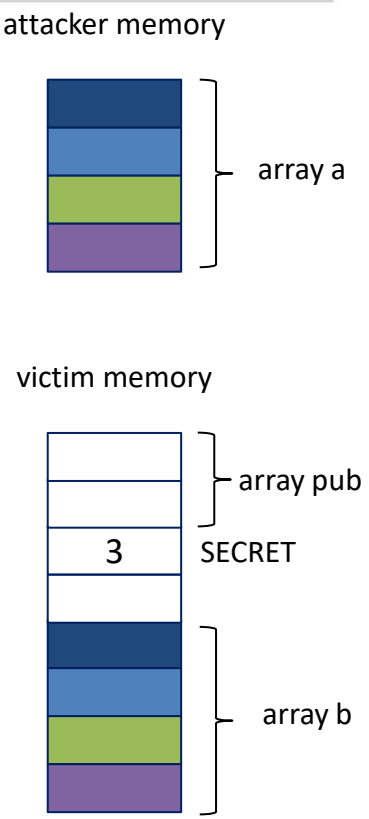
A simple example of a transient execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



A simple example of a transient execution attack

attacker code

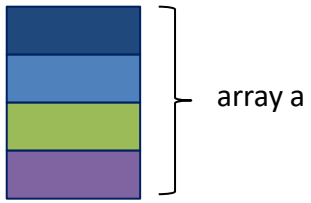
```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

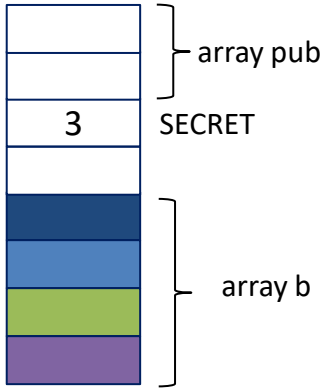
```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```

Branch predictor learns that usually then branch is taken

attacker memory



victim memory



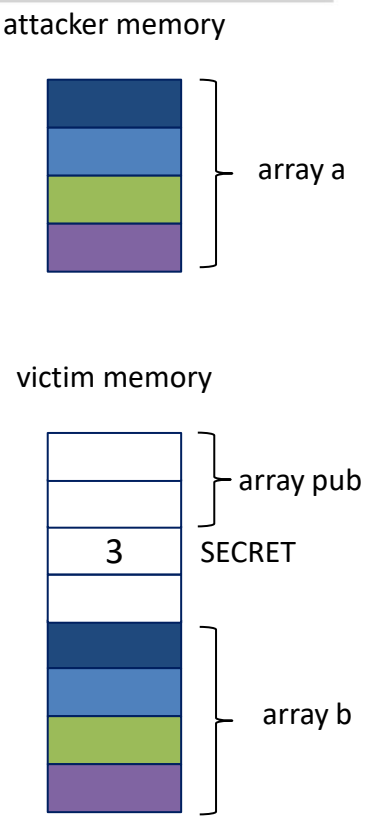
A simple example of a transient execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



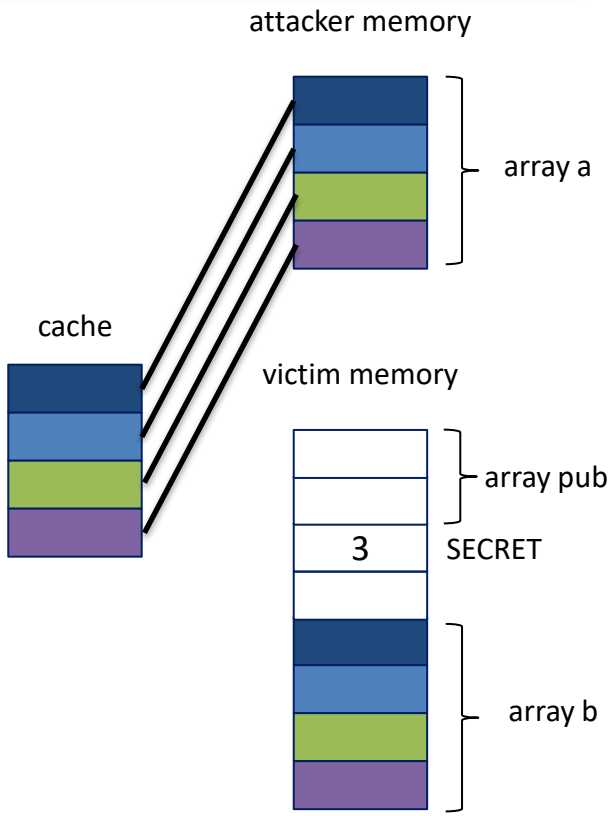
A simple example of a transient execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



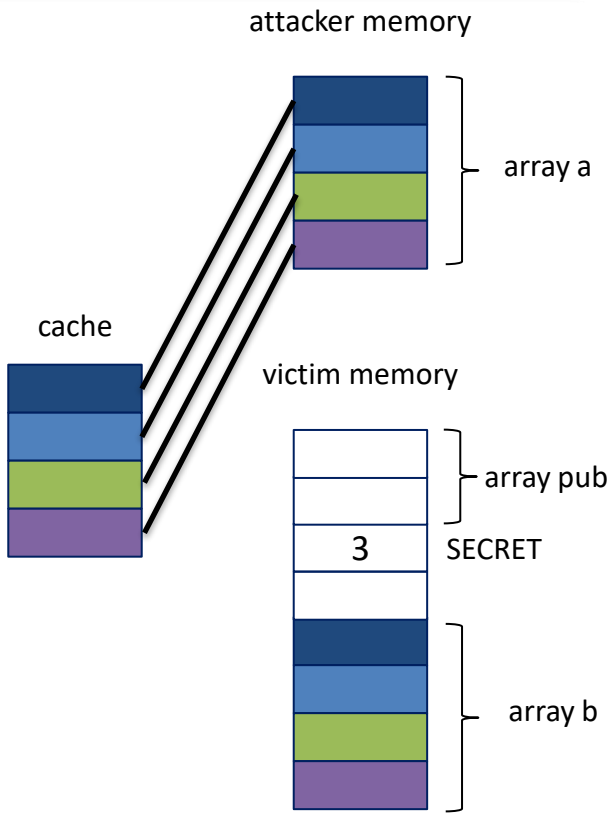
A simple example of a transient execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



A simple example of a transient execution attack

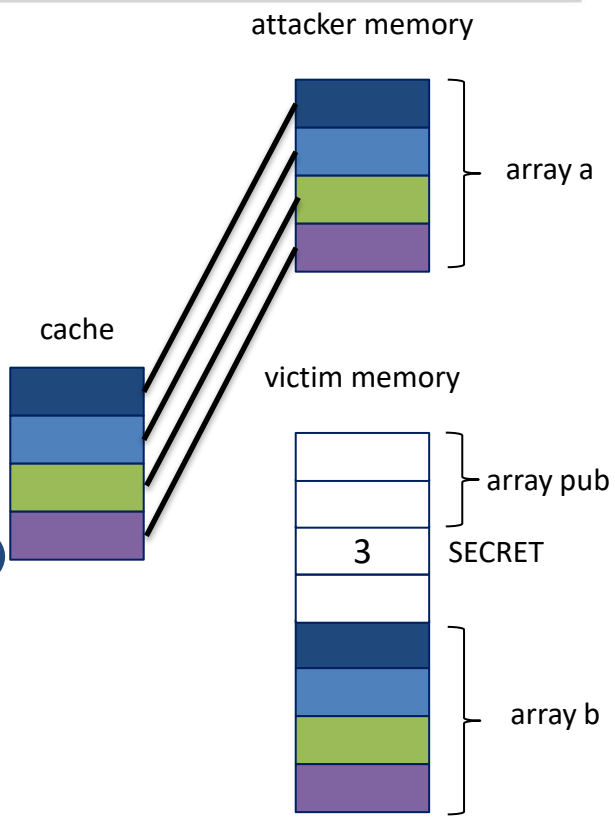
attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```

CPU speculatively executes the then branch



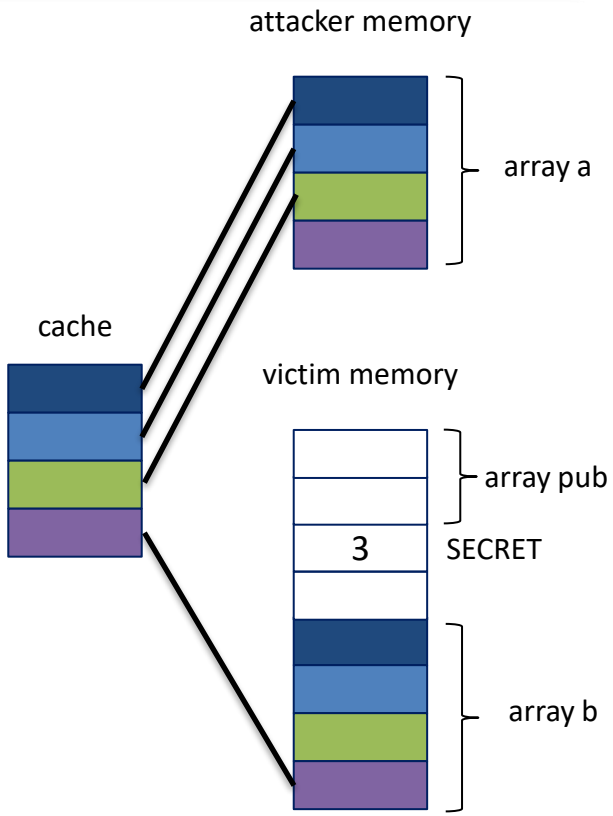
A simple example of a transient execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



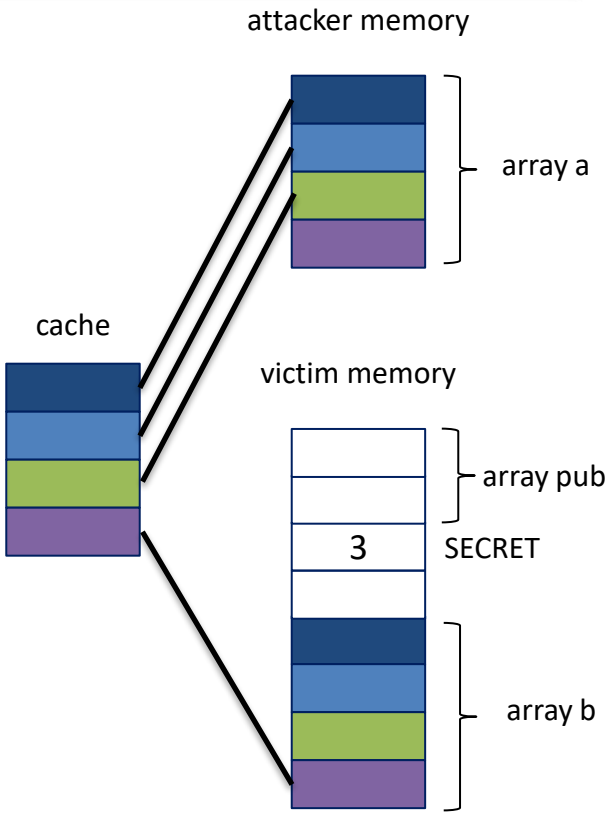
A simple example of a transient execution attack

attacker code

```
// train the branch predictor
process(0); process(0); …
// prime the cache
for (j=0; j<4; j++) z = a[j];
// attack!
process(size);
// measure access time to a[j] for all j
// slowest j is the SECRET
```

victim code

```
void process(int i) {
  int y;
  if (i < size) y = b[pub[i]];
}
```



Micro-architectural side-channel vulnerabilities

CyBOK

- This class of vulnerabilities is relatively new
  - Spectre and Meltdown were disclosed only in 2018
- The community is still building up a deeper understanding
  - Other kinds of attack techniques?
  - Should this be addressed in hardware or in software?

Conclusions

We have discussed three important categories of vulnerabilities that enable specific attack techniques against software.

The Software security KA document discusses more.

In addition, the document discusses how to protect software by preventing or detecting vulnerabilities, or by mitigating the effects of their exploitation.