End-to-end security for web applications

Stronger application isolation mechanisms for Tearless: why and how?

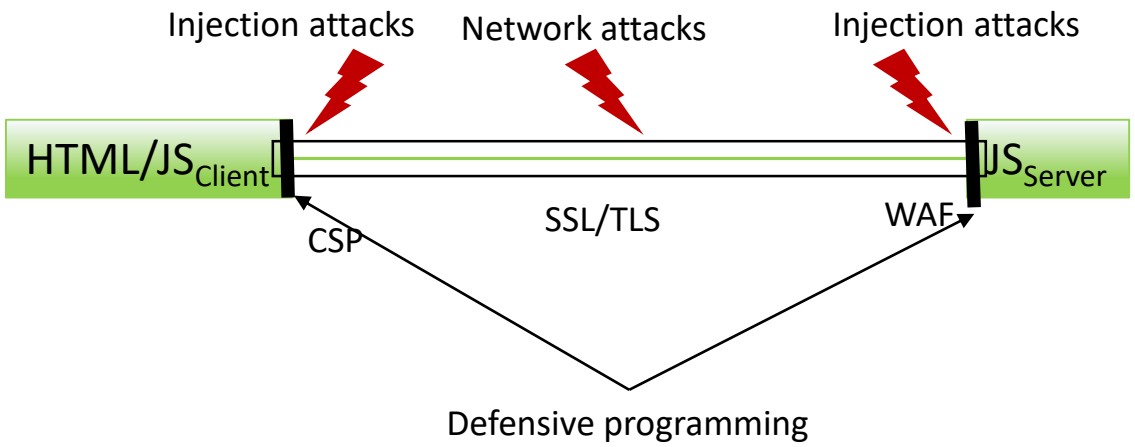
Frank Piessens

Tearless KU LEUVEN Vrije Universiteit Brussel

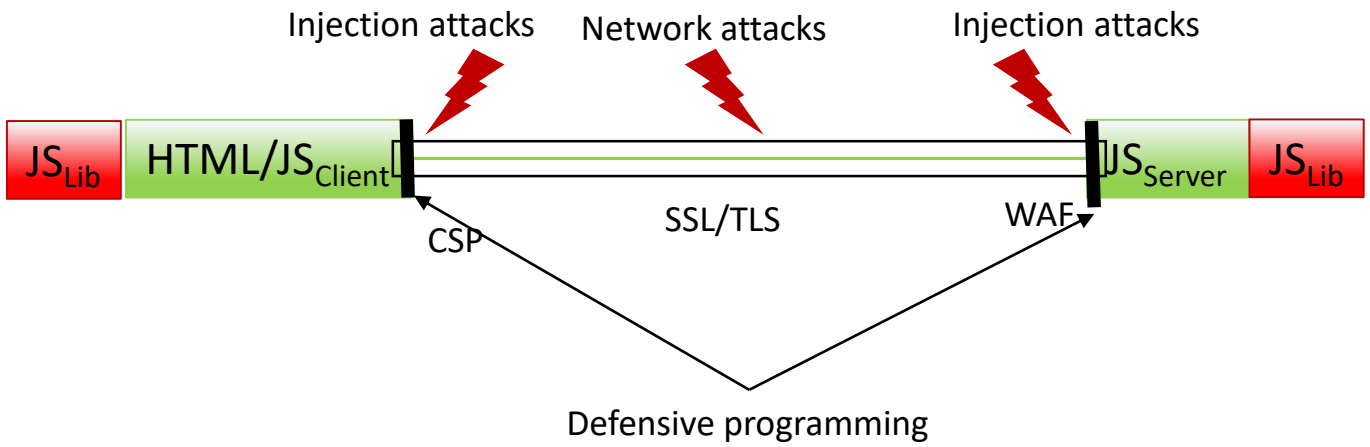
Introduction: a simplified view of a web app

Introduction: a simplified view of a web app

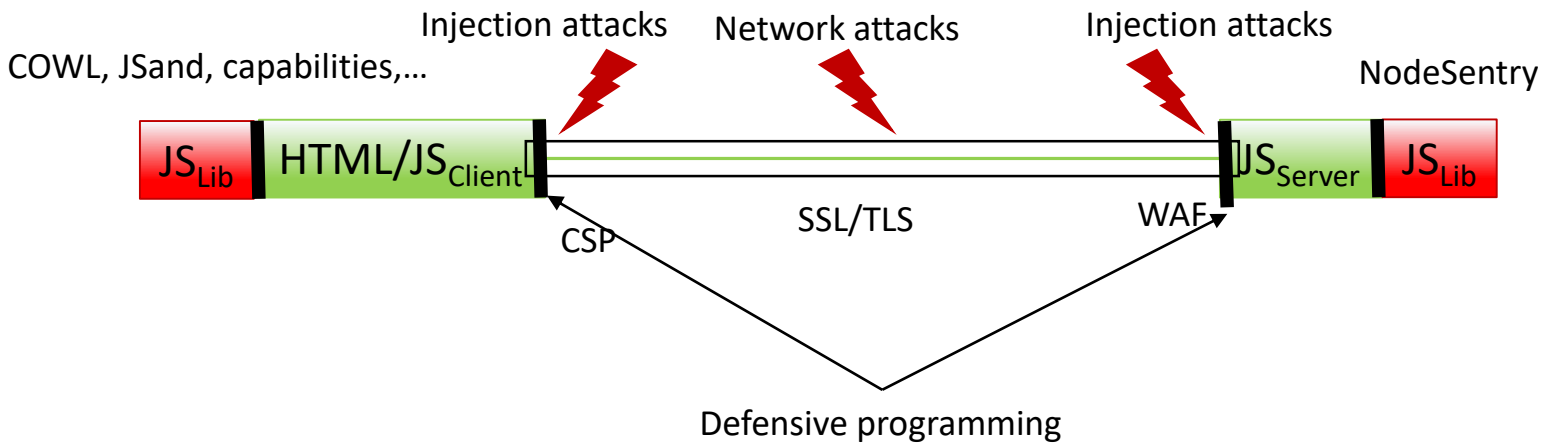
Introduction: a simplified view of a web app



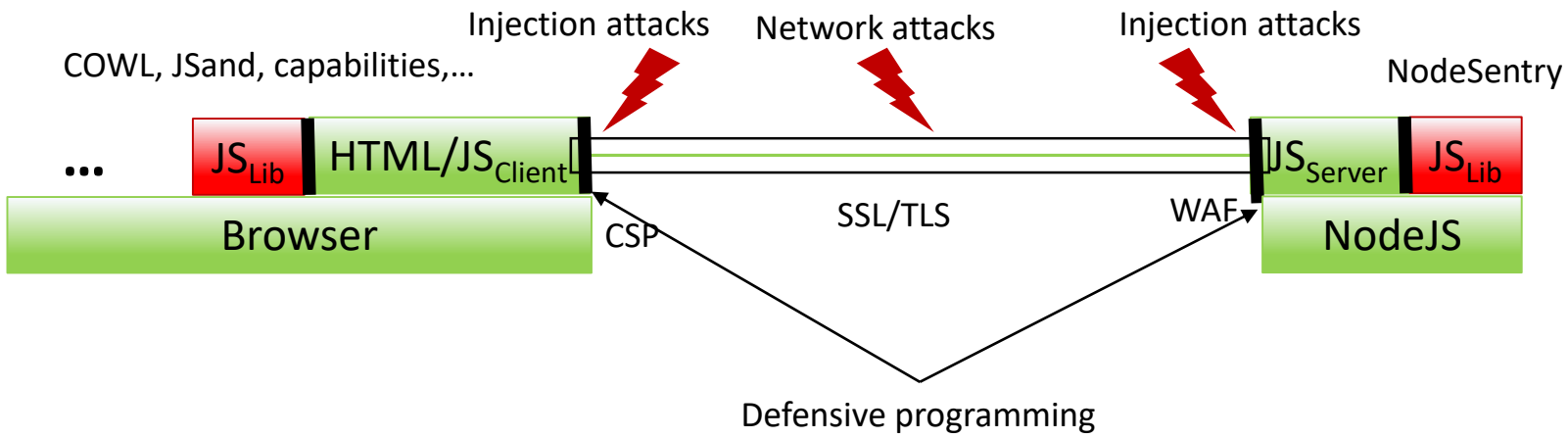
Introduction: a simplified view of a web app



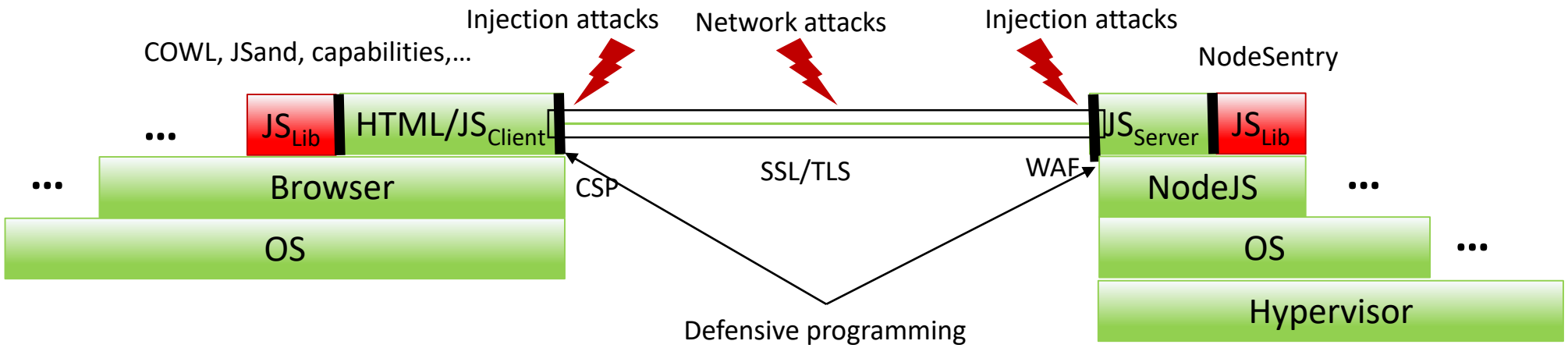
Introduction: a simplified view of a web app



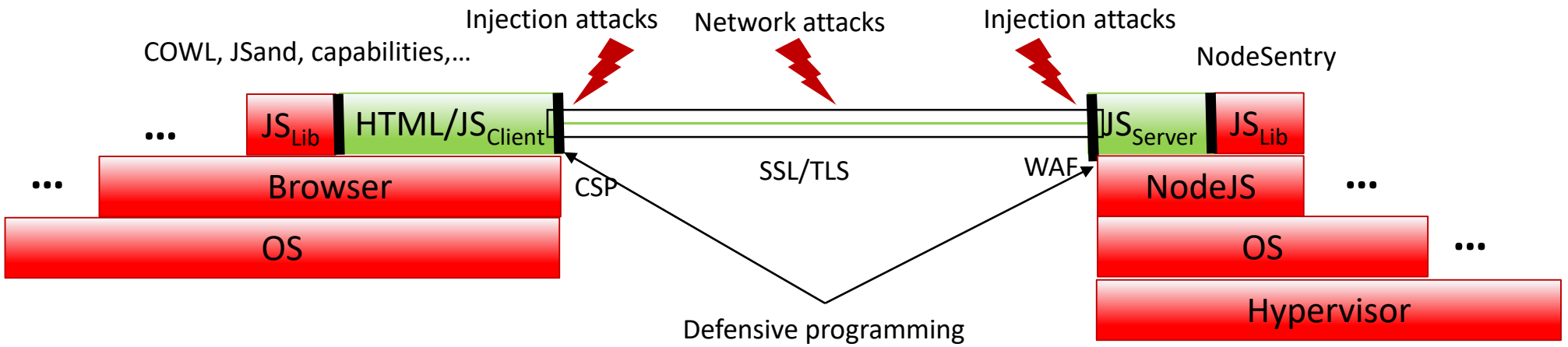
Introduction: a simplified view of a web app



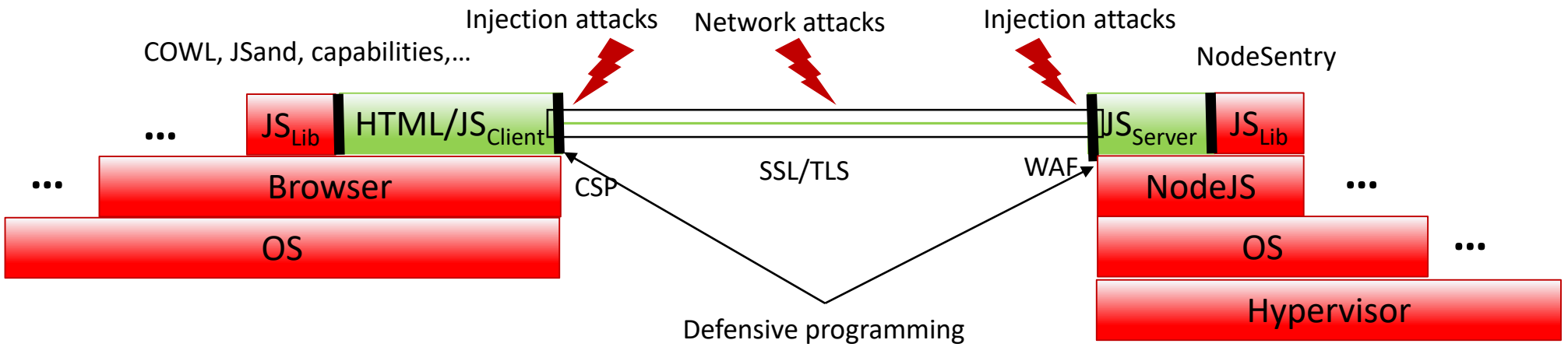
Introduction: a simplified view of a web app



Introduction: a simplified view of a web app



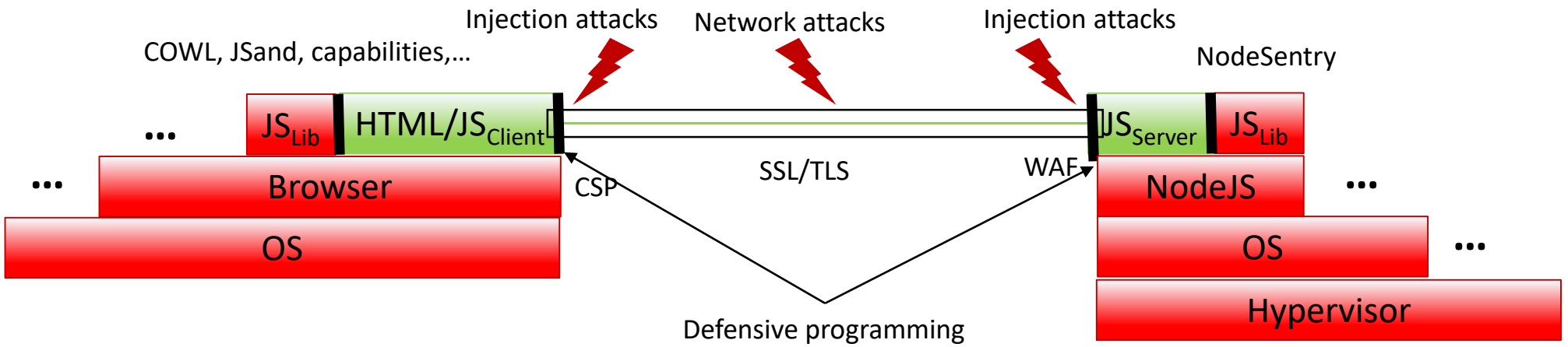
Introduction: a simplified view of a web app



- Man in the browser attacks
- Memory scanning attacks against POS terminals
- Kernel level malware
- …

- Stealing of credential databases
- Defacements
- Malicious cloud operators
- …

Introduction: a simplified view of a web app



- Man in the browser attacks
- Memory scanning attacks against POS terminals
- Kernel level malware
- …

- Stealing of credential databases
- Defacements
- Malicious cloud operators
- …

+ supporting offline operation by migrating server code and data to client!

Introduction: a simplified view of a web app

Conclusion:

We need stronger isolation mechanisms that protect against lower layers

- Man in the bro
- Memory scann
- Kernel level ma
- …

bases

- …

+ supporting offline operation by migrating server code and data to client!

Overview

- Protected module architectures (PMAs)
  - How can we provide protection against lower layers?
- Illustrating the use of PMAs in Tearless
- Research plan
- Conclusions/discussion

Overview

- Protected module architectures (PMAs)
  - How can we provide protection against lower layers?
- Illustrating the use of PMAs in Tearless
- Research plan
- Conclusions/discussion

Brief recap of the run-time state of programs

- Source code is compiled to machine code (JIT or ahead)
- Each function can be compiled separately
- Control flow through the program is tracked by the *call stack*
- Program data is stored:
  - In the *heap* for malloced data
  - In the stack for local variables
  - In the static data segment for global variables

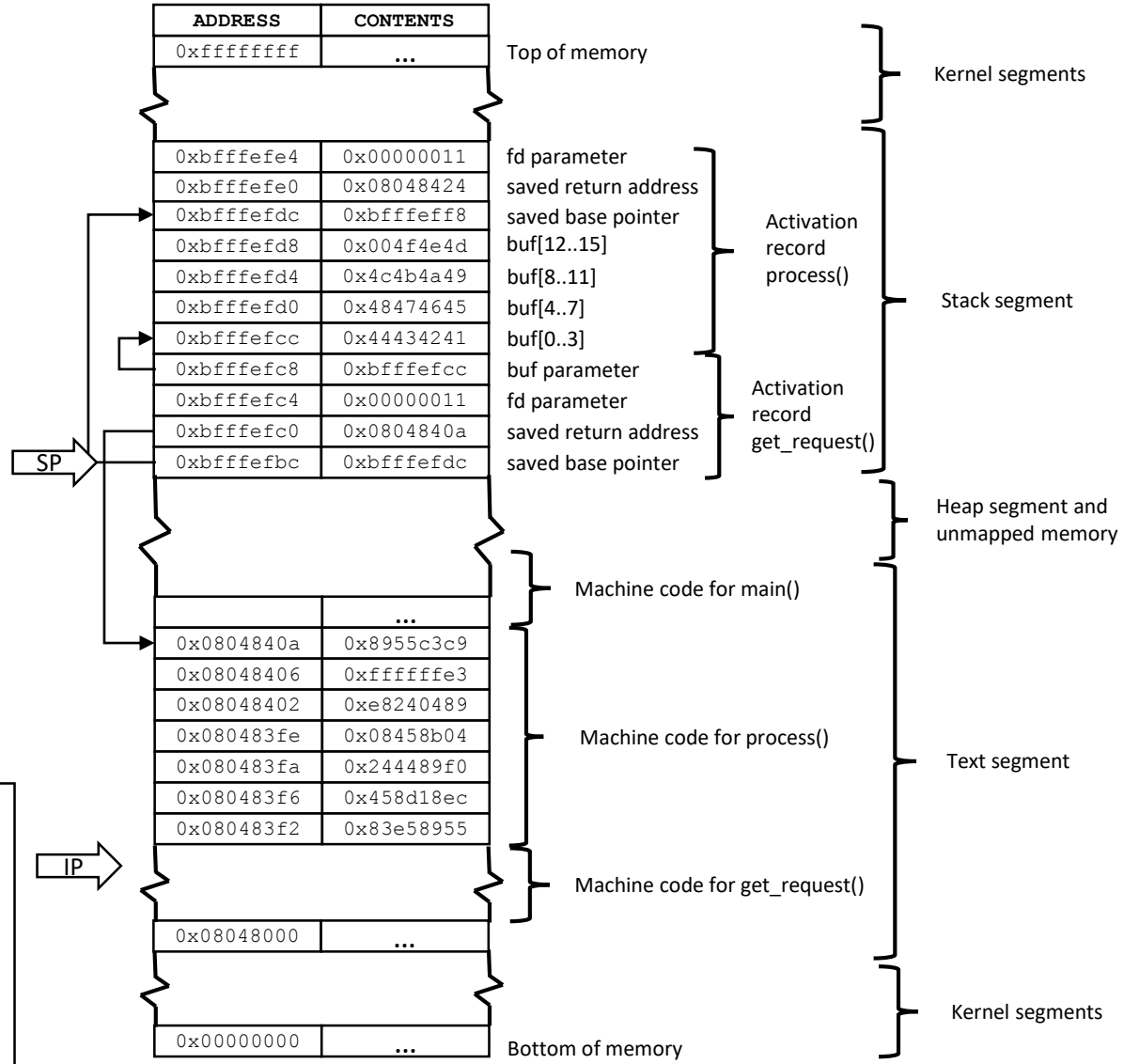
Detailed picture of the run-time state of a program

```
void get_request(int fd, char buf[]) {
  read(fd,buf,16);
}

void process(int fd) {
  char buf[16];
  get_request(fd,buf);
  // Process the request (code not shown)
}
void main() {
  int fd;
  // Initialize server, wait for a connection
  // Accept connection, with file descriptor fd
  // Finally, process the request:
  process(fd);
}
```

(a) Program source code

```
55              push   %ebp              ; save base pointer
89 e5           mov    %esp,%ebp         ; set new base pointer
83 ec 18        sub    $0x18,%esp        ; allocate stack record
8d 45 f0        lea    -0x10(%ebp),%eax  ; put buf in %eax
89 44 24 04     mov    %eax,0x4(%esp)    ; and push on the stack
8b 45 08        mov    0x8(%ebp),%eax    ; put fd parameter in %eax
89 04 24        mov    %eax,(%esp)       ; and push on the stack
e8 e3 ff ff ff  call   0x80483ed         ; call get_request
c9              leave                    ; deallocate stack frame
c3              ret                      ; return
```

(b) Machine code for process() function



(c) Run-time machine state on entering get_request()

Attack possibilities from lower layers

secret.h

```
int get_secret(int provided_pin)
```

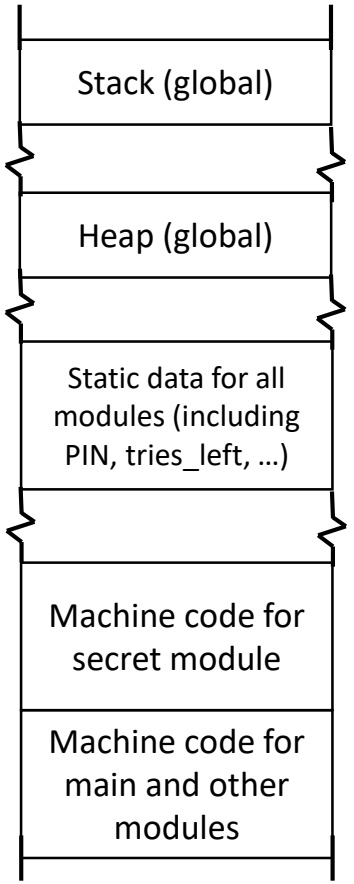
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
      tries_left = 3;
     return secret;}
   else { tries_left-- ; return 0; }; }
else return 0; }
```

(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
…
}
```

(b) Other modules of the program



(c) Run-time memory contents

- Arbitrary changes to code and data
  - Memory scraping
  - Violating licensing restrictions
  - …

Protected modules

secret.h

```
int get_secret(int provided_pin)
```

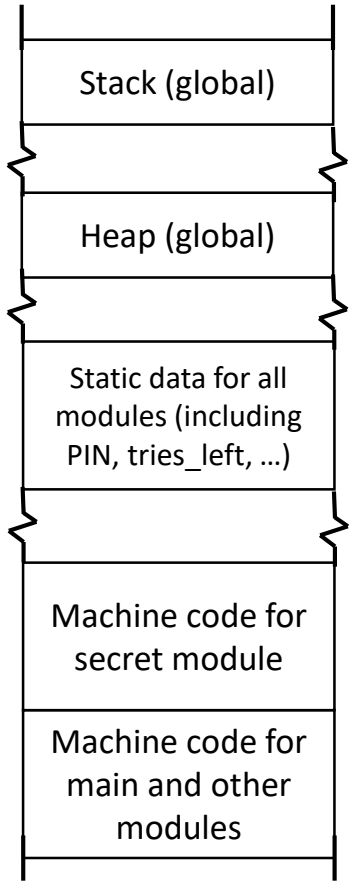
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
      tries_left = 3;
     return secret;}
   else { tries_left-- ; return 0; }; }
else return 0; }
```

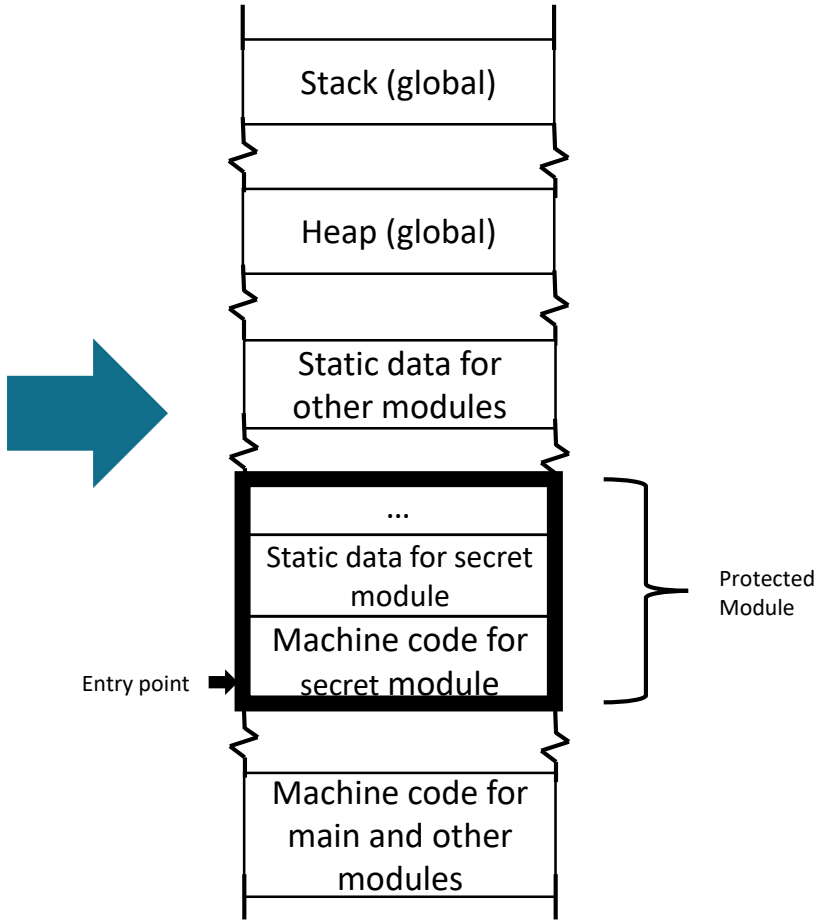
(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
…
}
```

(b) Other modules of the program



(c) Run-time memory contents



Protected module architectures

- Several conceptually similar systems were developed relatively independently
  - The pioneering system: Flicker
    - Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, Hiroshi Isozaki: Flicker: an execution infrastructure for TCB minimization. EuroSys 2008
  - Many variants
    - Intel Software Guard Extensions (Intel SGX)
    - Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, Frank Piessens: Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. USENIX Security 2013
    - Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, Vijay Varadharajan: TrustLite: a security architecture for tiny embedded devices. EuroSys 2014
    - ….

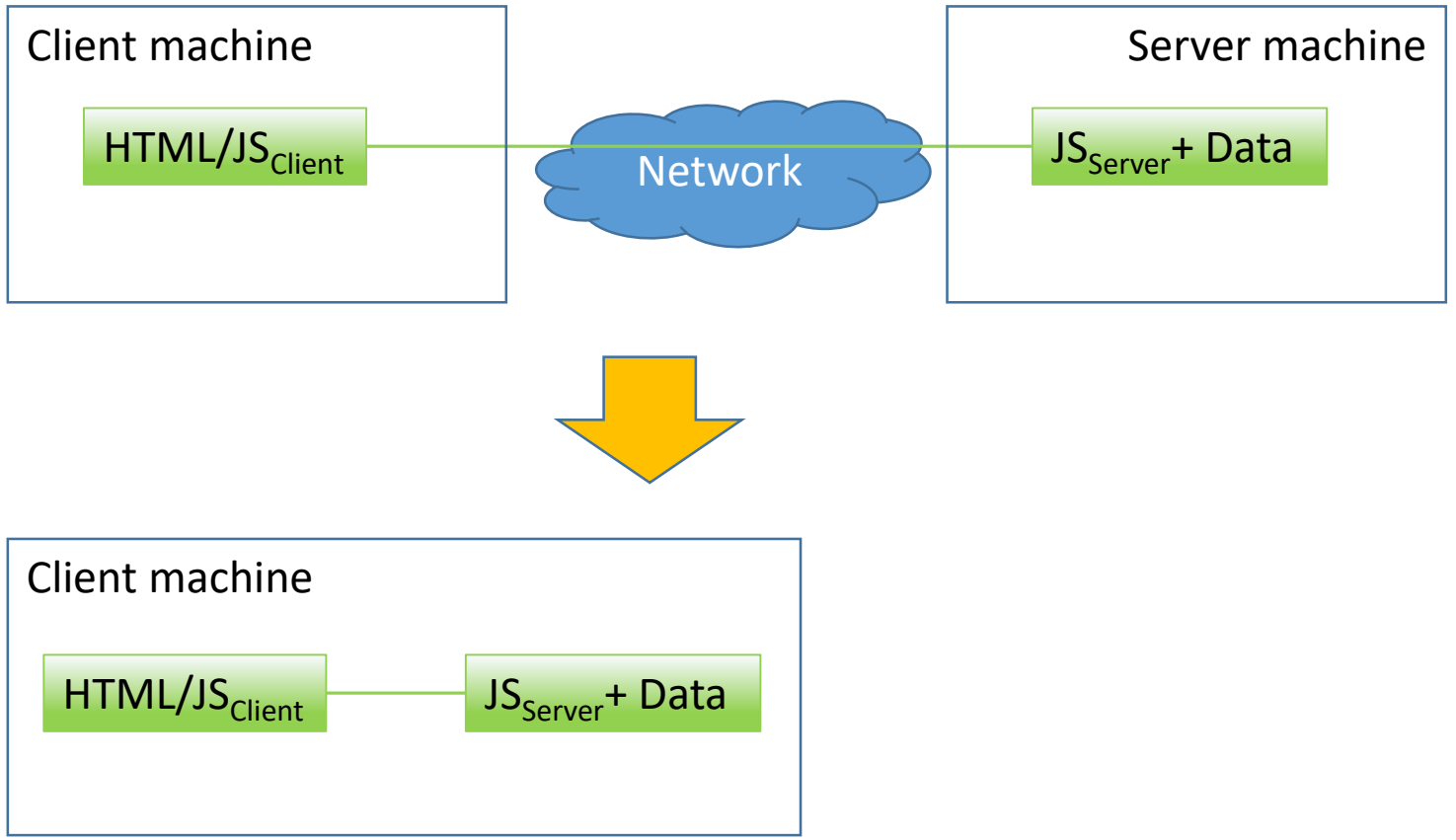
One-slide summary

- PMAs give you
  - A way to load code on a system in an isolated container (*enclave*) that **no software on the system** can peek into or tamper with
  - A way to **remotely** check that such a correctly initialized enclave is present
  - A way to set up **a secure connection** with such an enclave

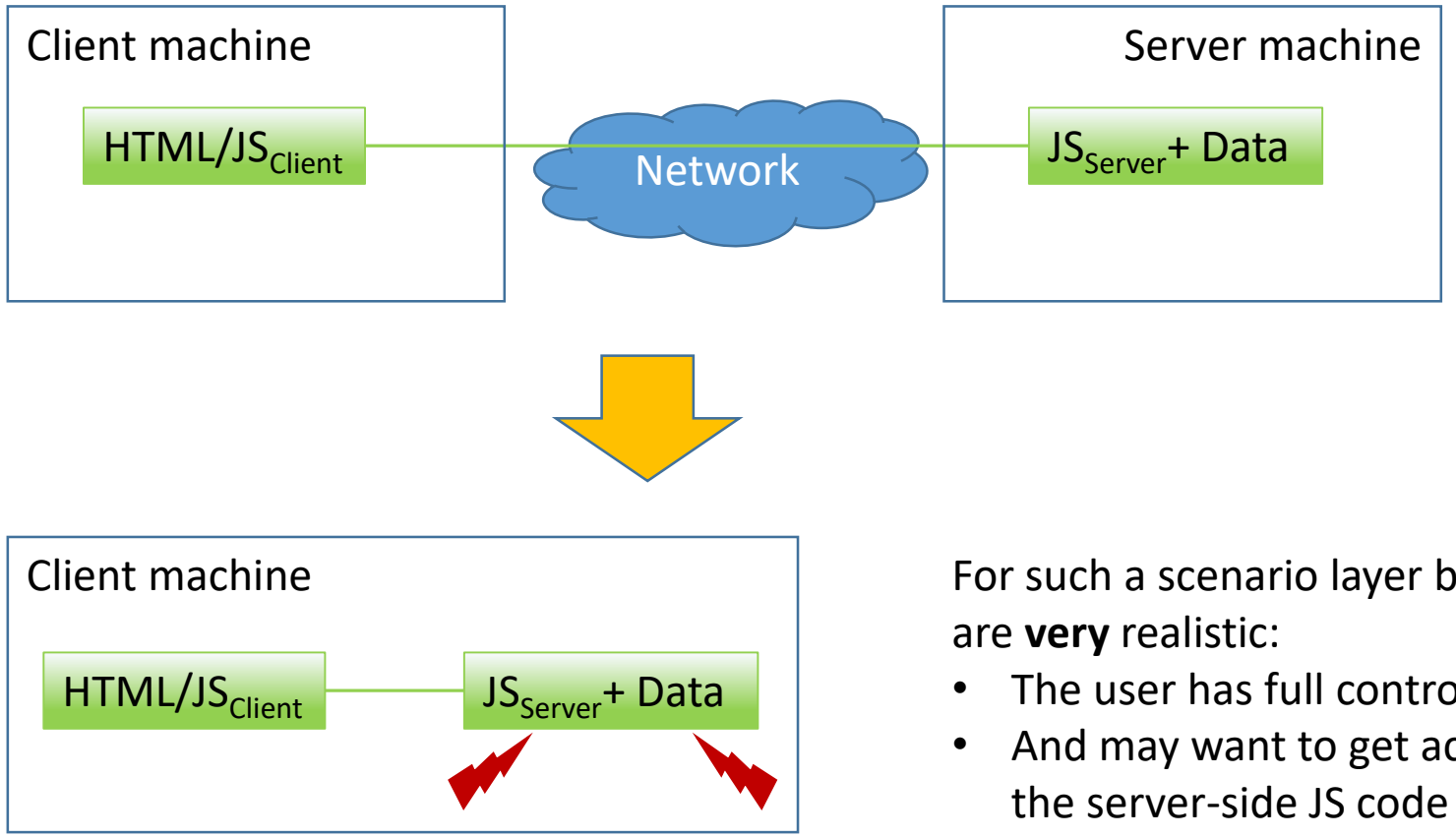
Overview

- Protected module architectures (PMAs)
  - How can we provide protection against lower layers?
- Illustrating the use of PMAs in Tearless
- Research plan
- Conclusions/discussion

Secure offline operation



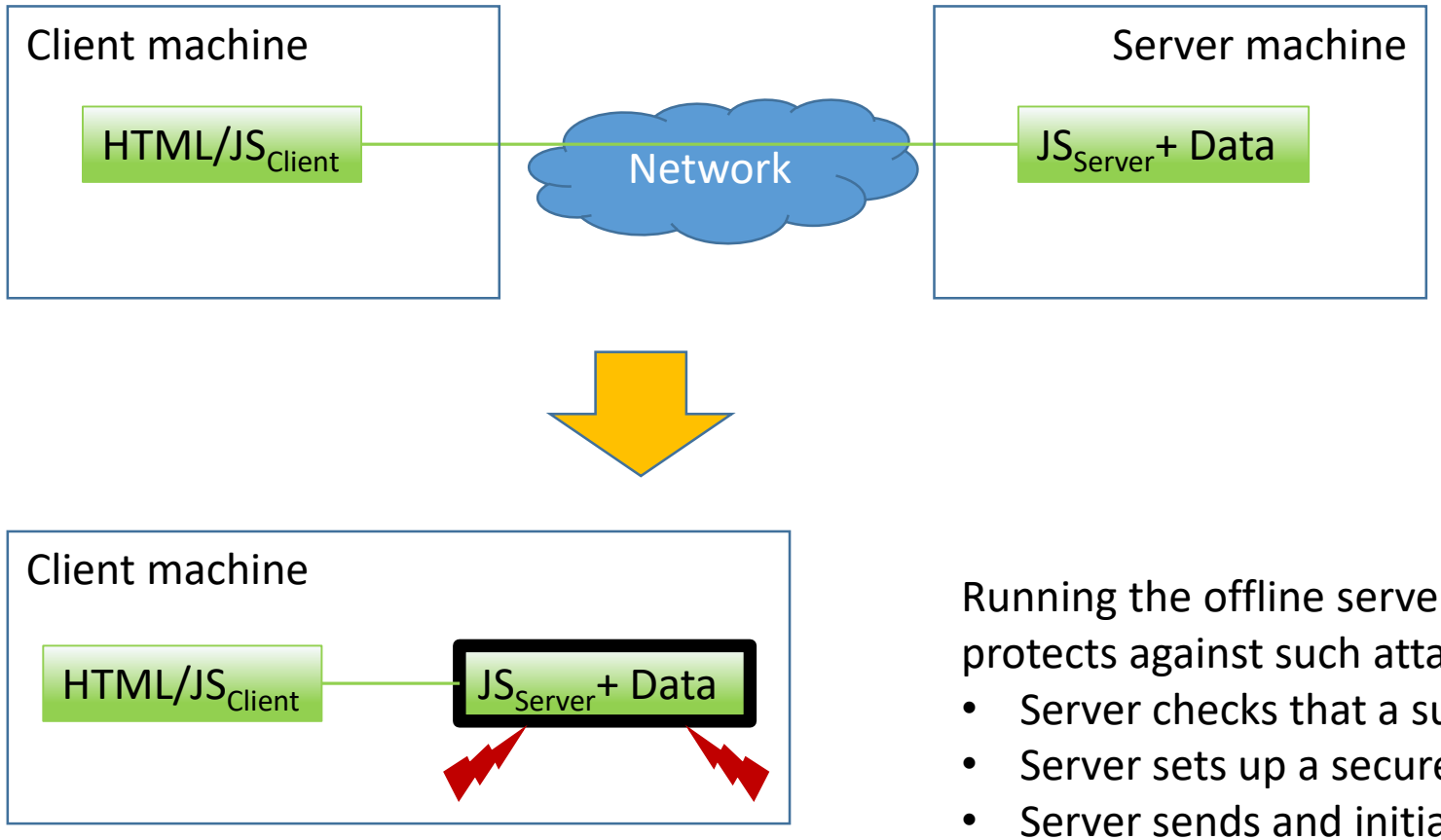
Secure offline operation



For such a scenario layer below attacks are **very** realistic:

- The user has full control over the client machine
- And may want to get access to data shielded by the server-side JS code

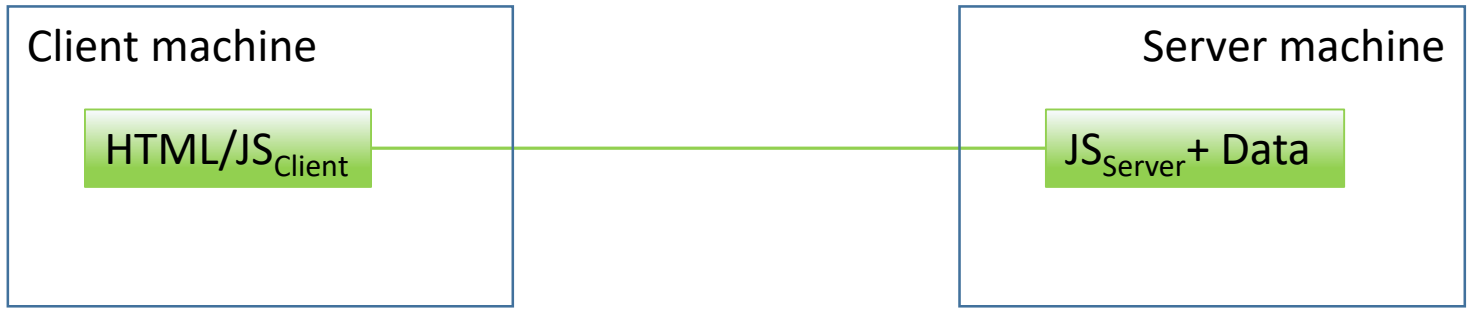
Secure offline operation



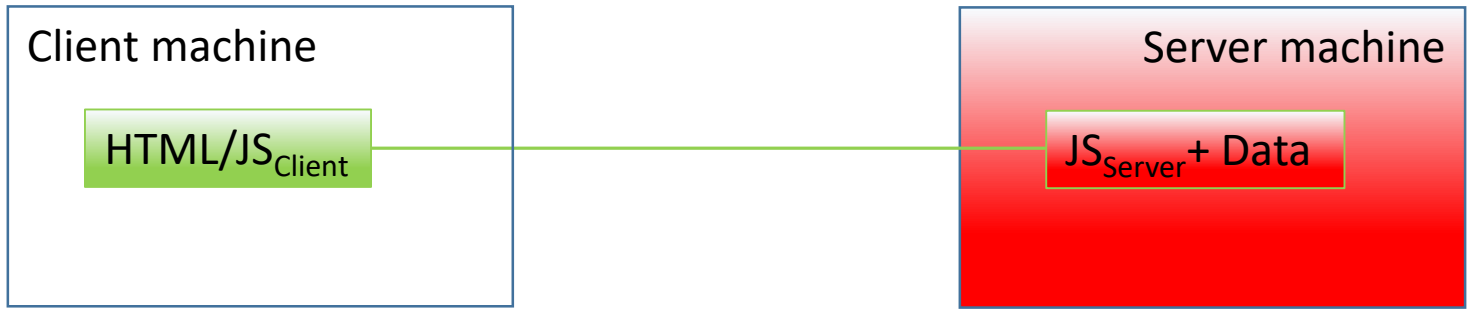
Running the offline server code in an SGX enclave protects against such attacks:

- Server checks that a suitable JS runtime is loaded
- Server sets up a secure channel to the enclave
- Server sends and initializes “offline server code”

Protecting server-side sensitive data

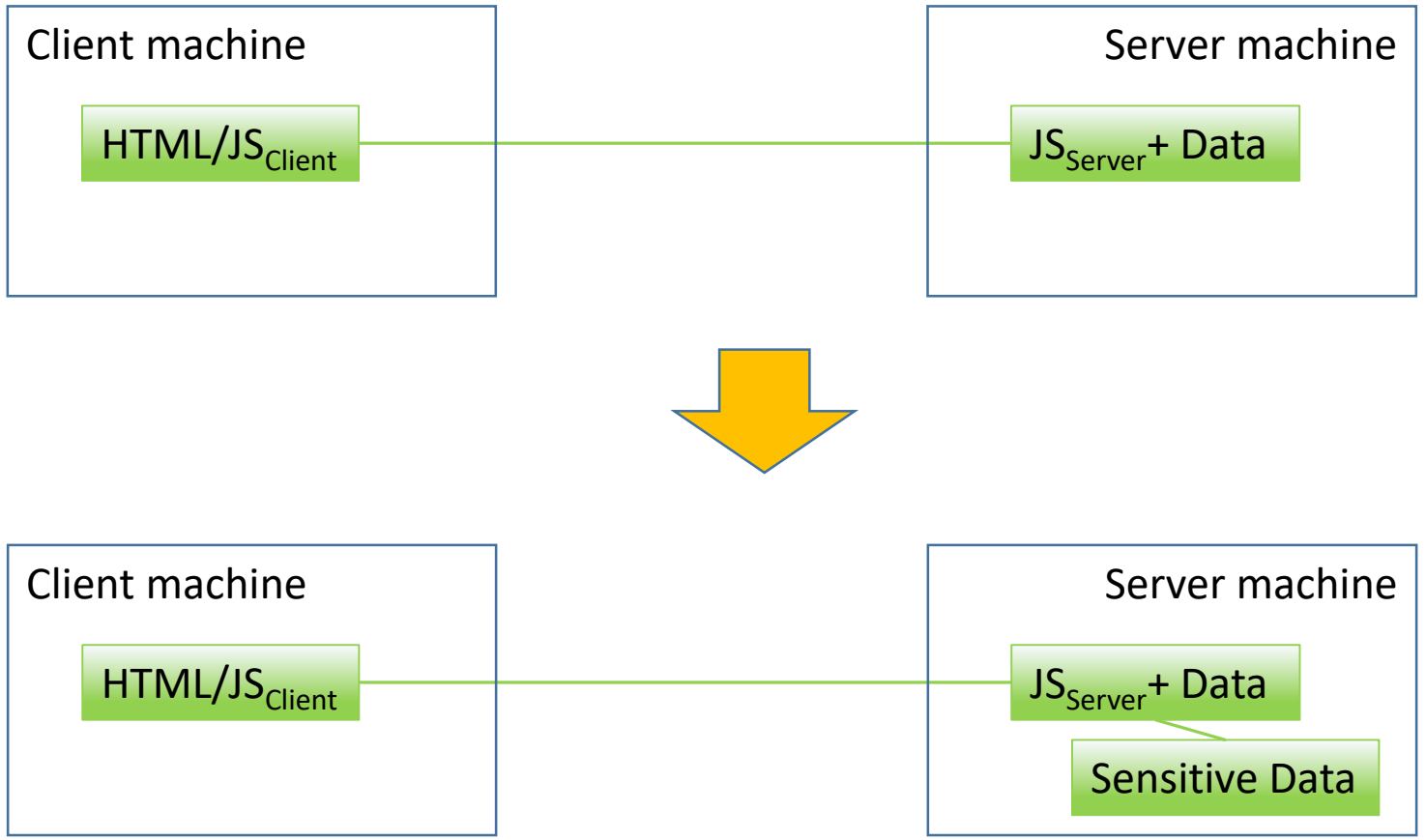


Protecting server-side sensitive data

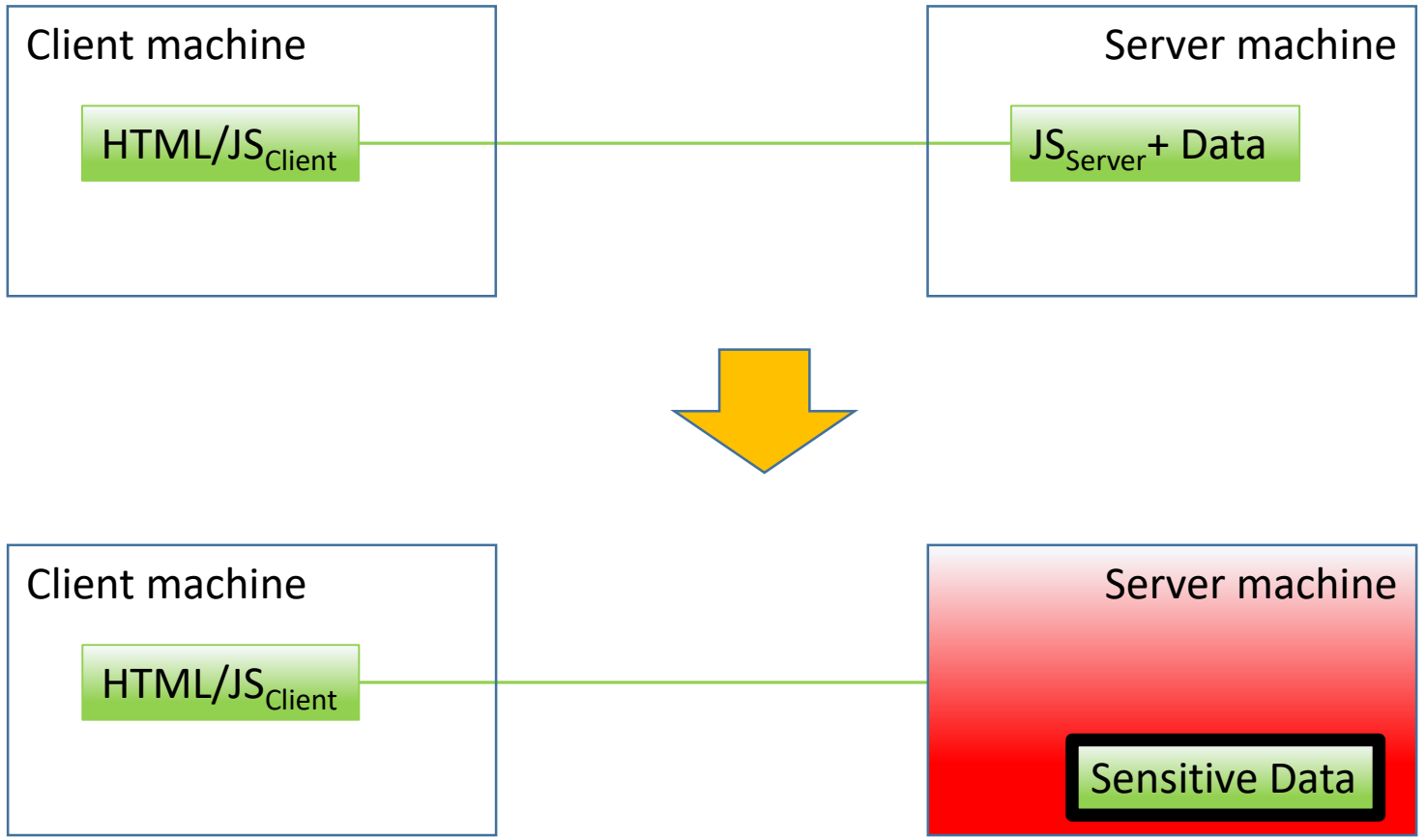


In case of server compromise
all server-side code and
data is compromised

Protecting server-side sensitive data



Protecting server-side sensitive data



Server-side SGX enclave protects most sensitive data

Overview

- Protected module architectures (PMAs)
  - How can we provide protection against lower layers?
- Illustrating the use of PMAs in Tearless
- Research plan
- Conclusions/discussion

Research plan (ongoing work)

- Understanding the isolation properties offered by Intel SGX to higher-level languages
- Handling persistence of SGX enclaves
- Porting JavaScript engines to an SGX enclave
- Server-side enclaves as an additional tier in a multi-tier system
- …

Overview

- Protected module architectures (PMAs)
  - How can we provide protection against lower layers?
- Illustrating the use of PMAs in Tearless
- Research plan
- Conclusions/discussion

Conclusions

- A realistic attacker model for multi-tier applications should include layer-below-attackers
- Hence the security track in Tearless is investigating protection mechanisms against such attackers
- Protected Module Architectures (and specifically Intel SGX) are a very promising protection mechanism

Questions / Comments / Discussion