Pointless Tainting?

Evaluating the Practicality of Pointer Tainting

Asia Slowinska, Herbert Bos
Vrije Universiteit Amsterdam

Why pointer tainting?

- Attacks

  Exploit low-level memory errors

  - Buffer overflows
  - Dangling pointers
  - Format strings

Why pointer tainting?

- **Attacks**
  Exploit low-level memory errors

Buffer overflows

Dangling pointers

Format strings

Control-diverting

Why pointer tainting?

- **Attacks**
  Exploit low-level memory errors

Buffer overflows

Dangling pointers

Format strings

Control-diverting

Non-control-diverting

Why pointer tainting?

- Attacks

  Exploit low-level memory errors

  - Buffer overflows
  - Dangling pointers
  - Format strings
    - Control-diverting
    - Non-control-diverting

- Keyloggers, etc.

  Installed by users or by the way of exploits

  - e.g., trojan

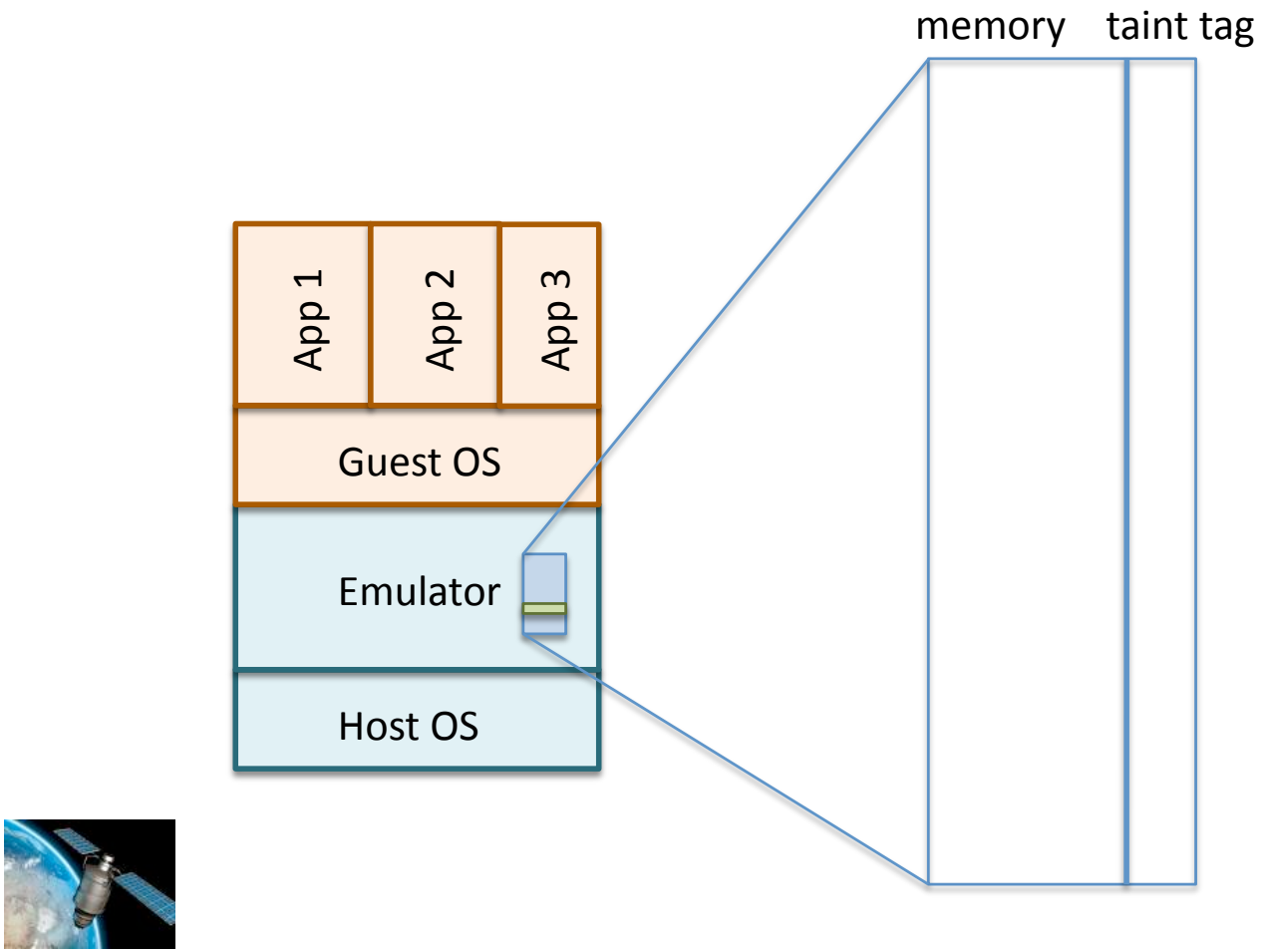
Why pointer tainting?

- Attacks
  - Exploit low-level memory errors
- Buffer overflows
- Dangling pointers
- Format strings
- Control-diverti
- Non-control-dive

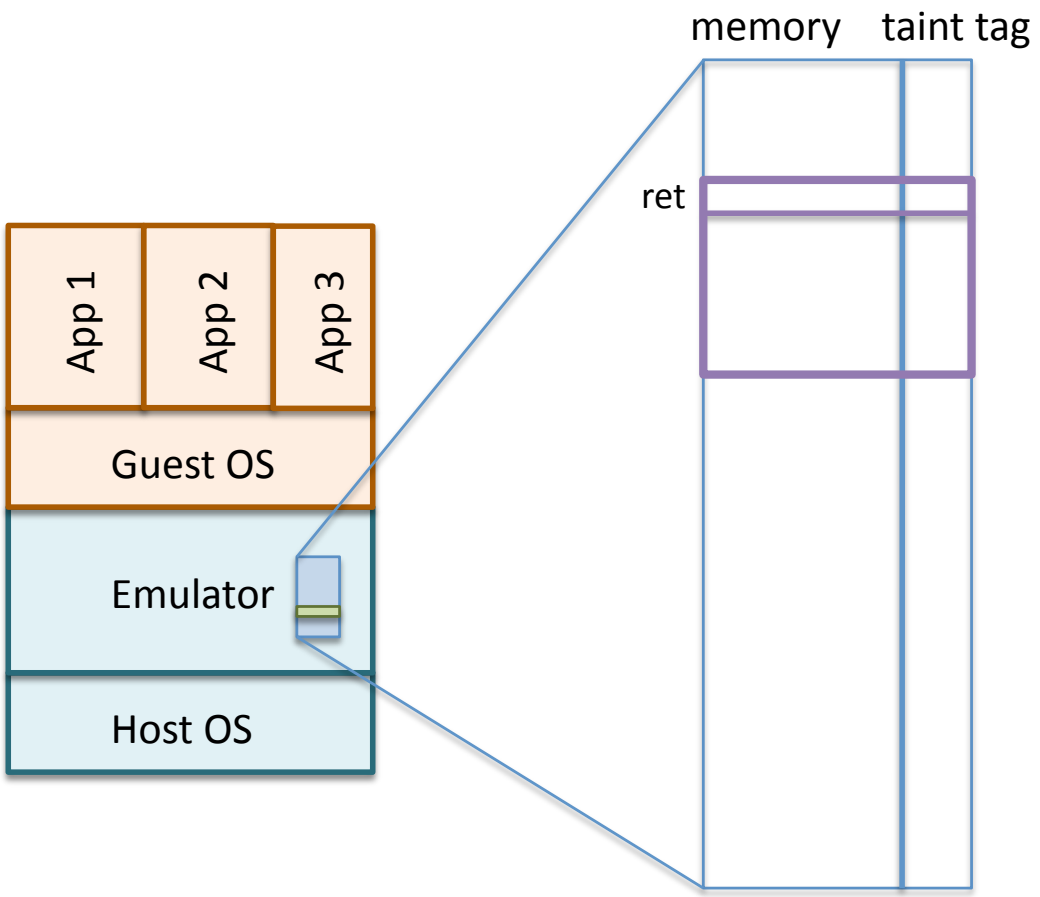
- Keyloggers, etc.
  - Installed by users or by the way of exploits
- e.g., trojan

- **Pointer tainting**
  - Capable of detecting
    - Memory corruption attacks
      - Both control- and non-control-diverting
    - Privacy-breaching malware
  - PROBLEMATIC

Basic tainting

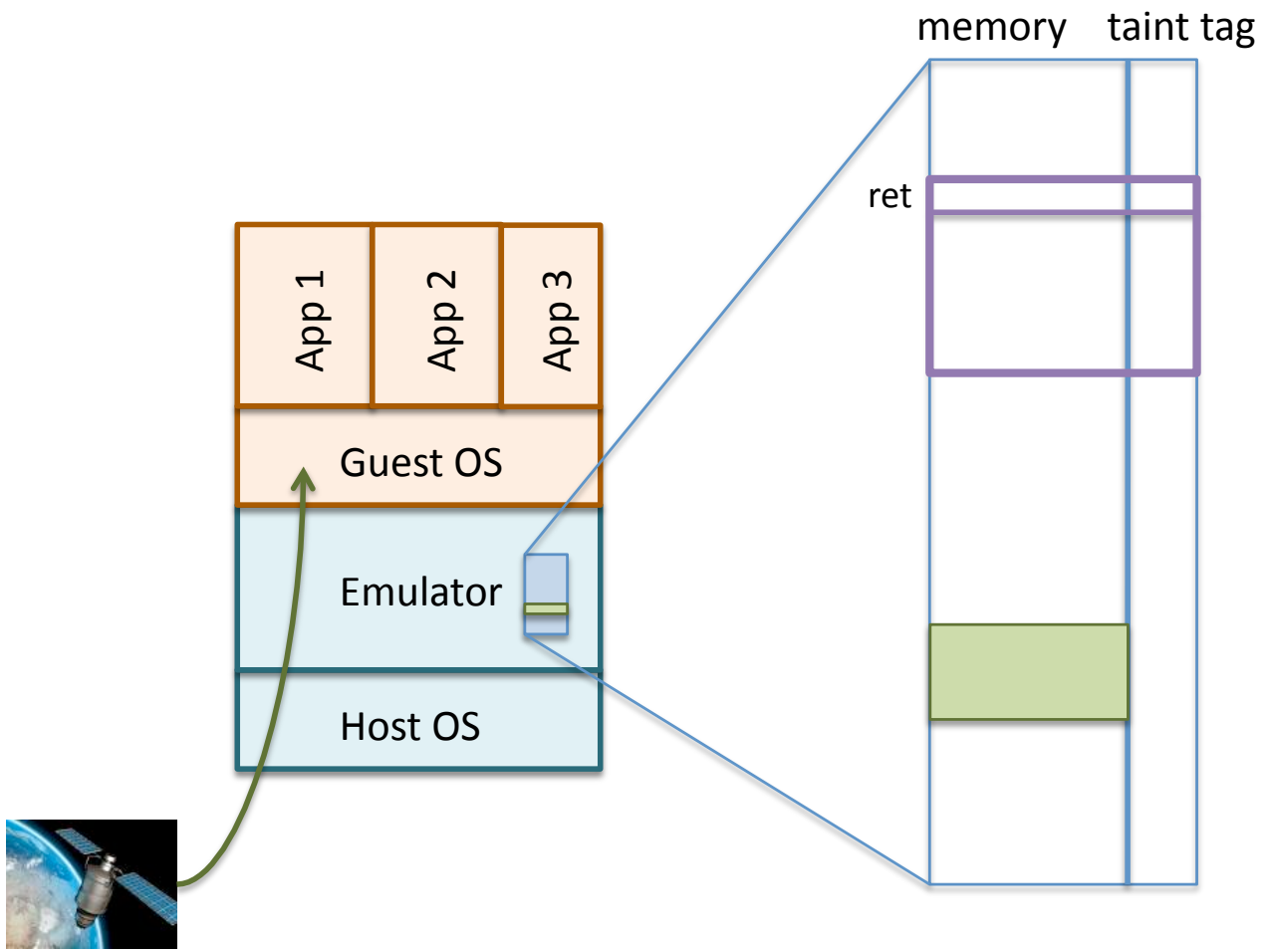


Basic tainting



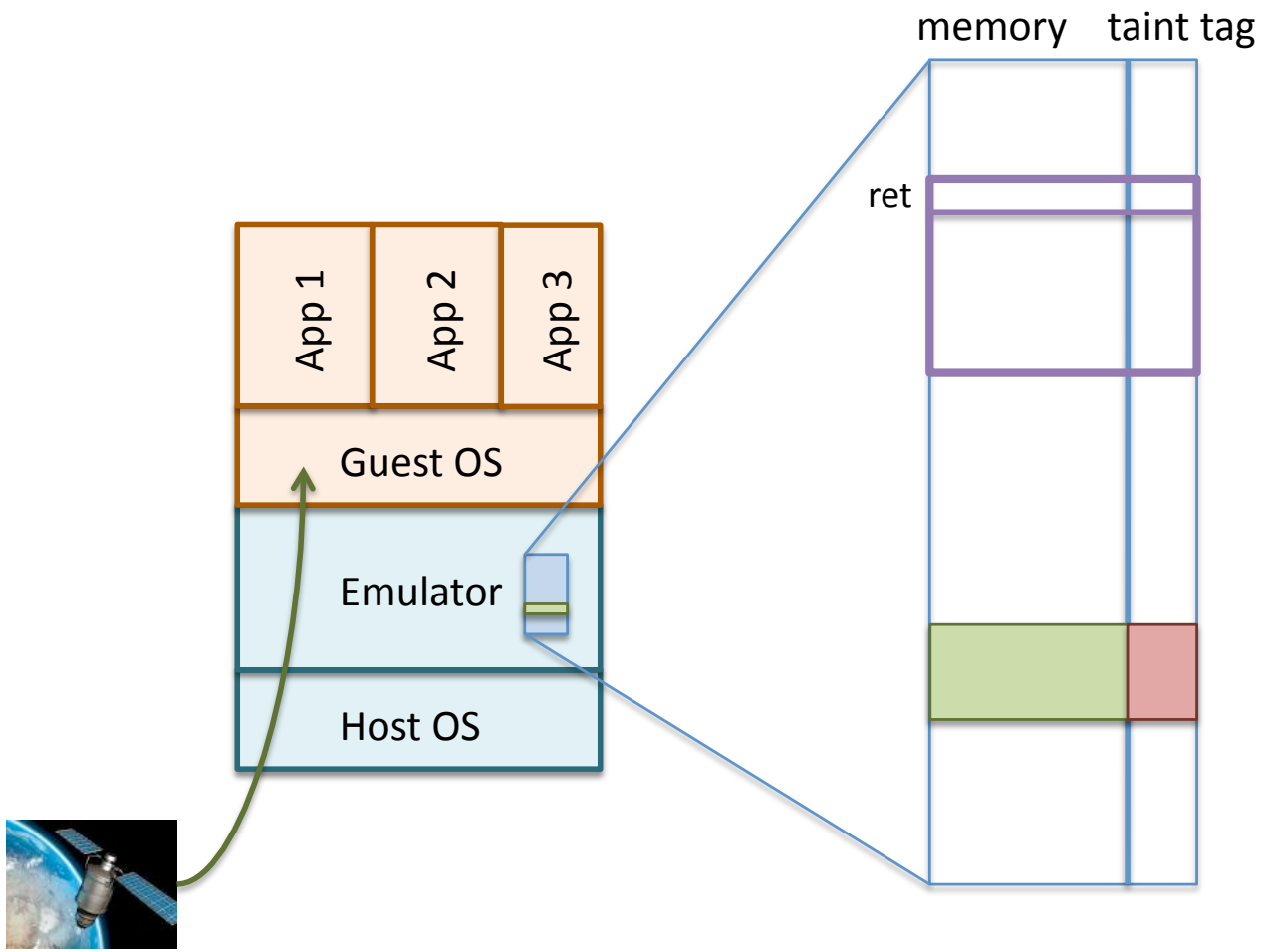
Basic tainting

1. Mark network data as tainted.

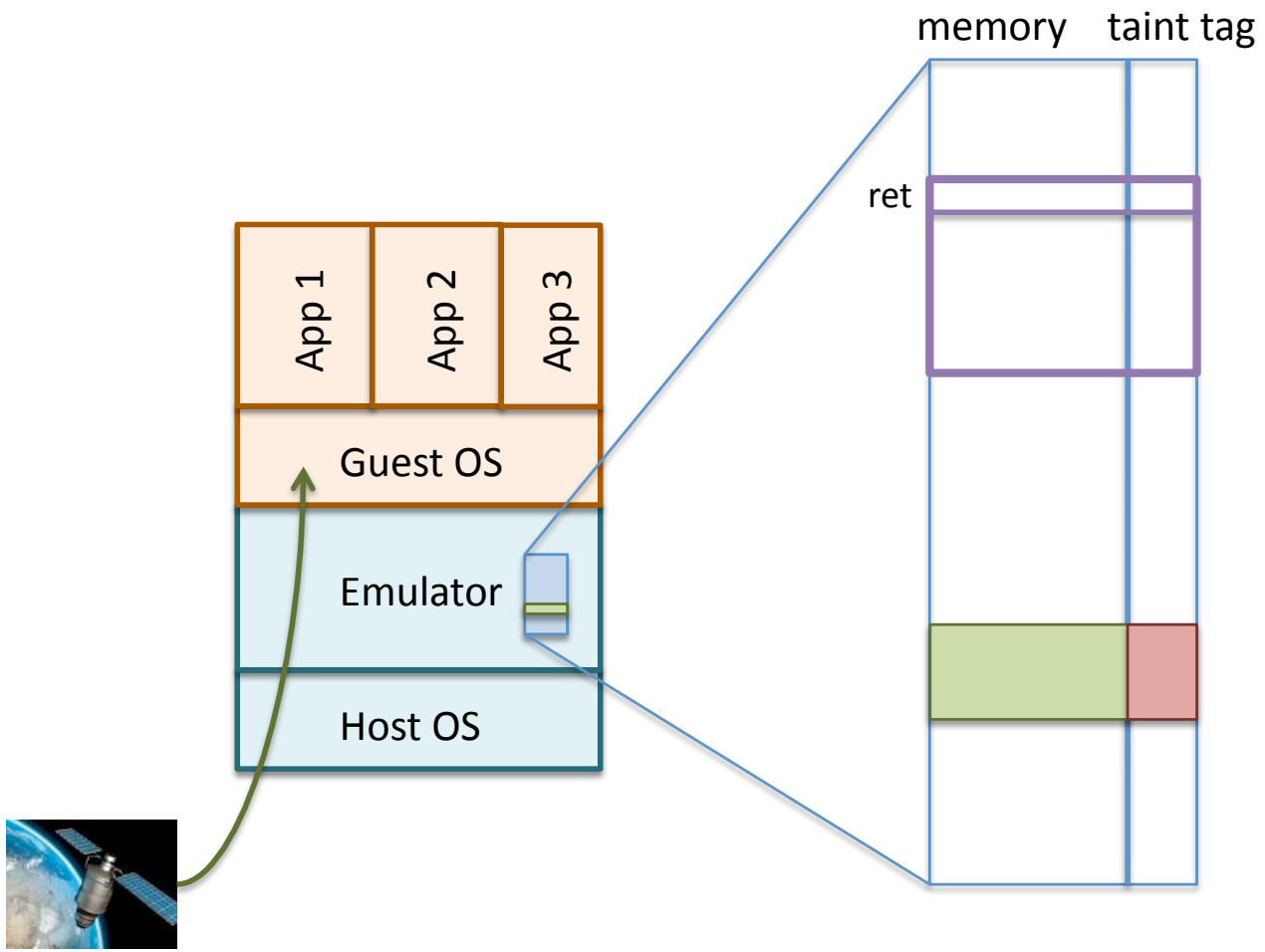


Basic tainting

1. Mark network data as tainted.

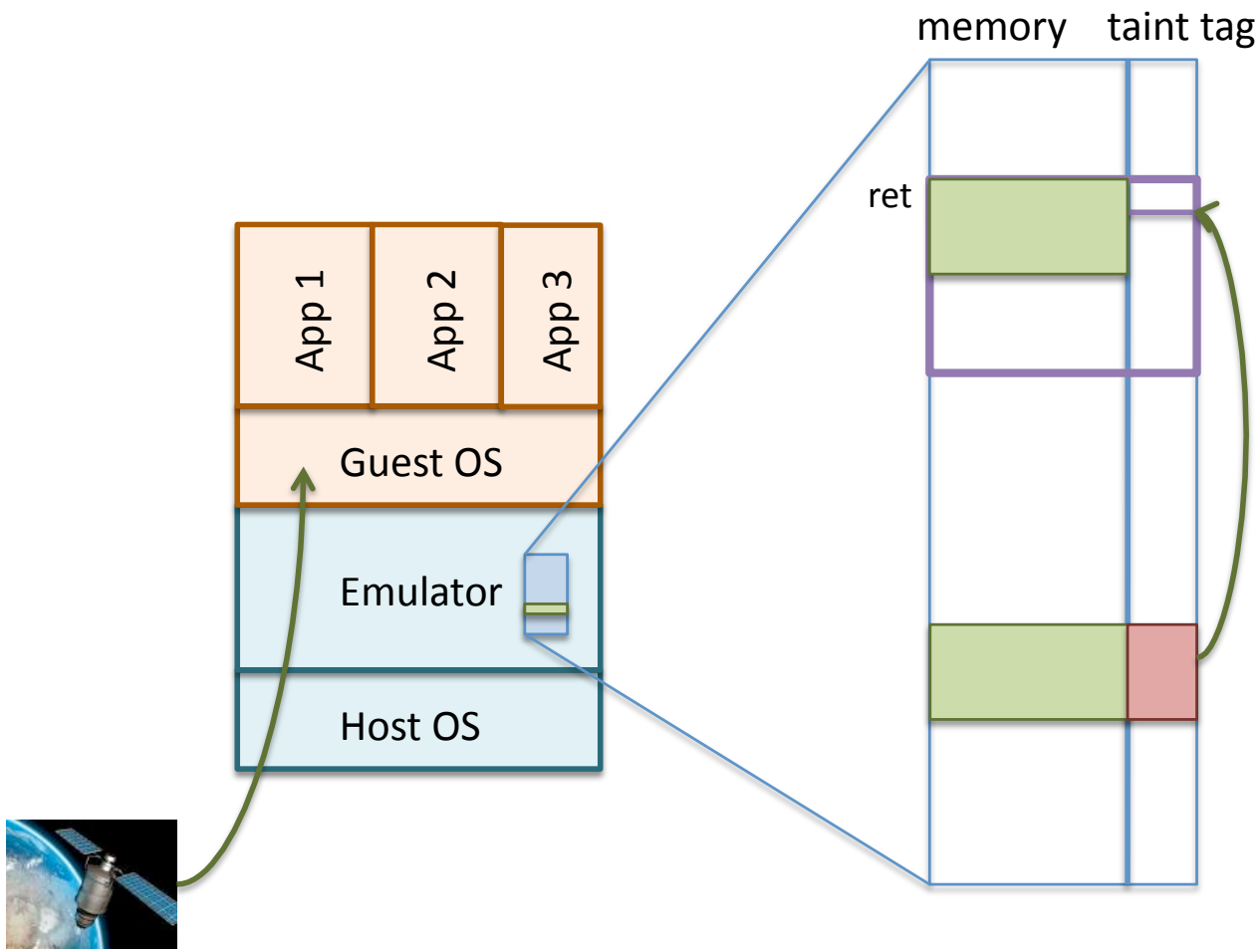


Basic tainting



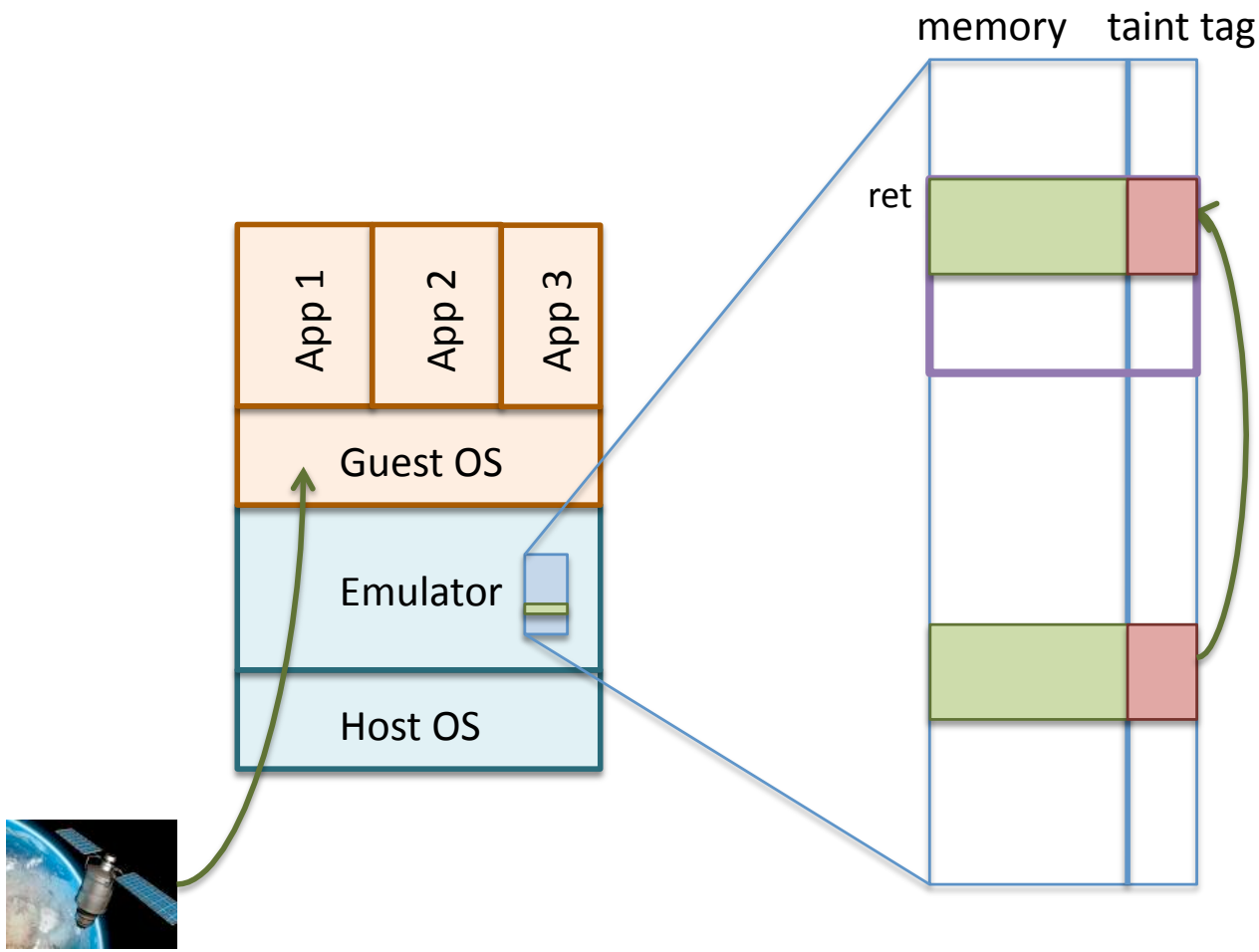
1. Mark network data as tainted.
2. Propagate taint through the OS.

Basic tainting



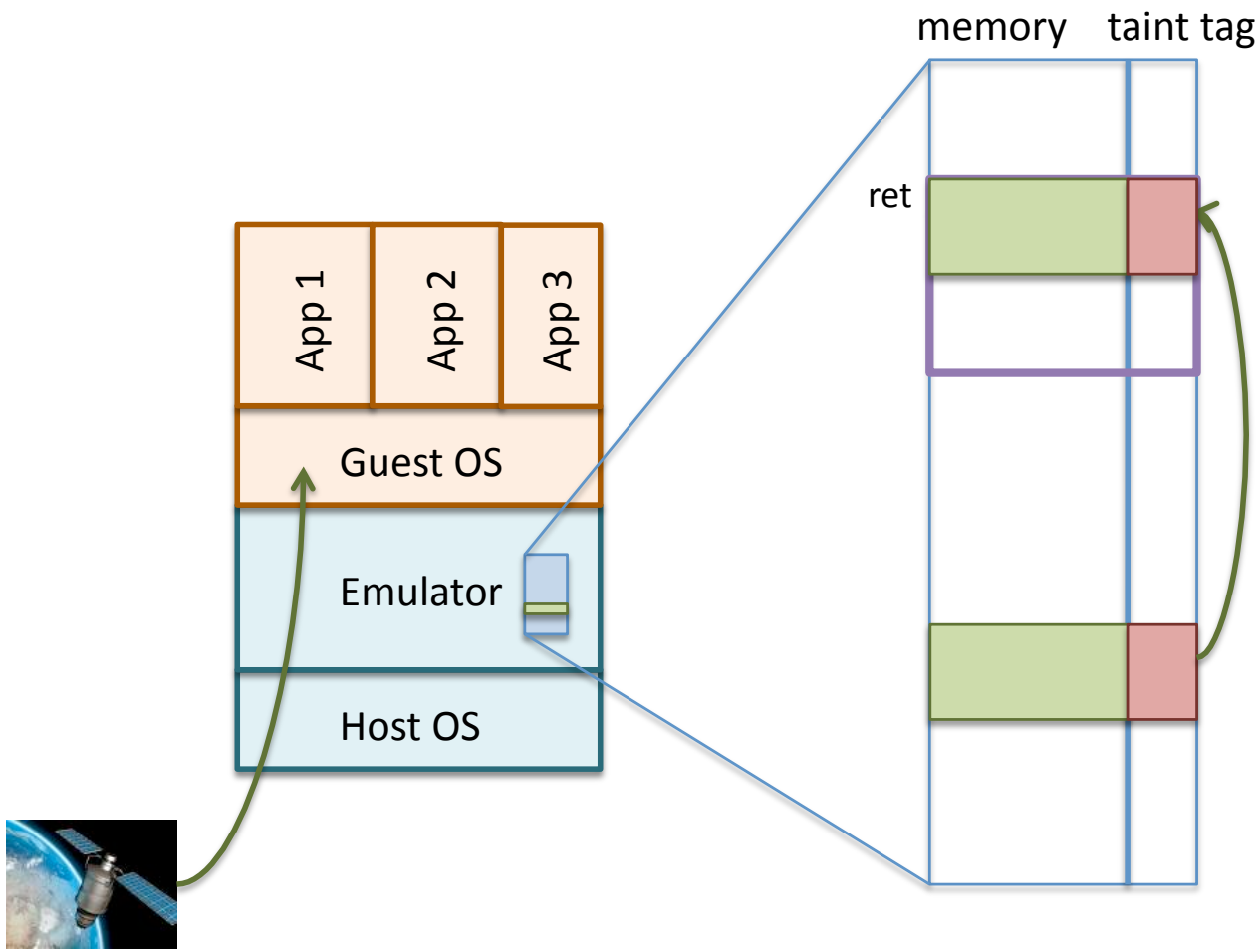
1. Mark network data as tainted.
2. Propagate taint through the OS.

Basic tainting



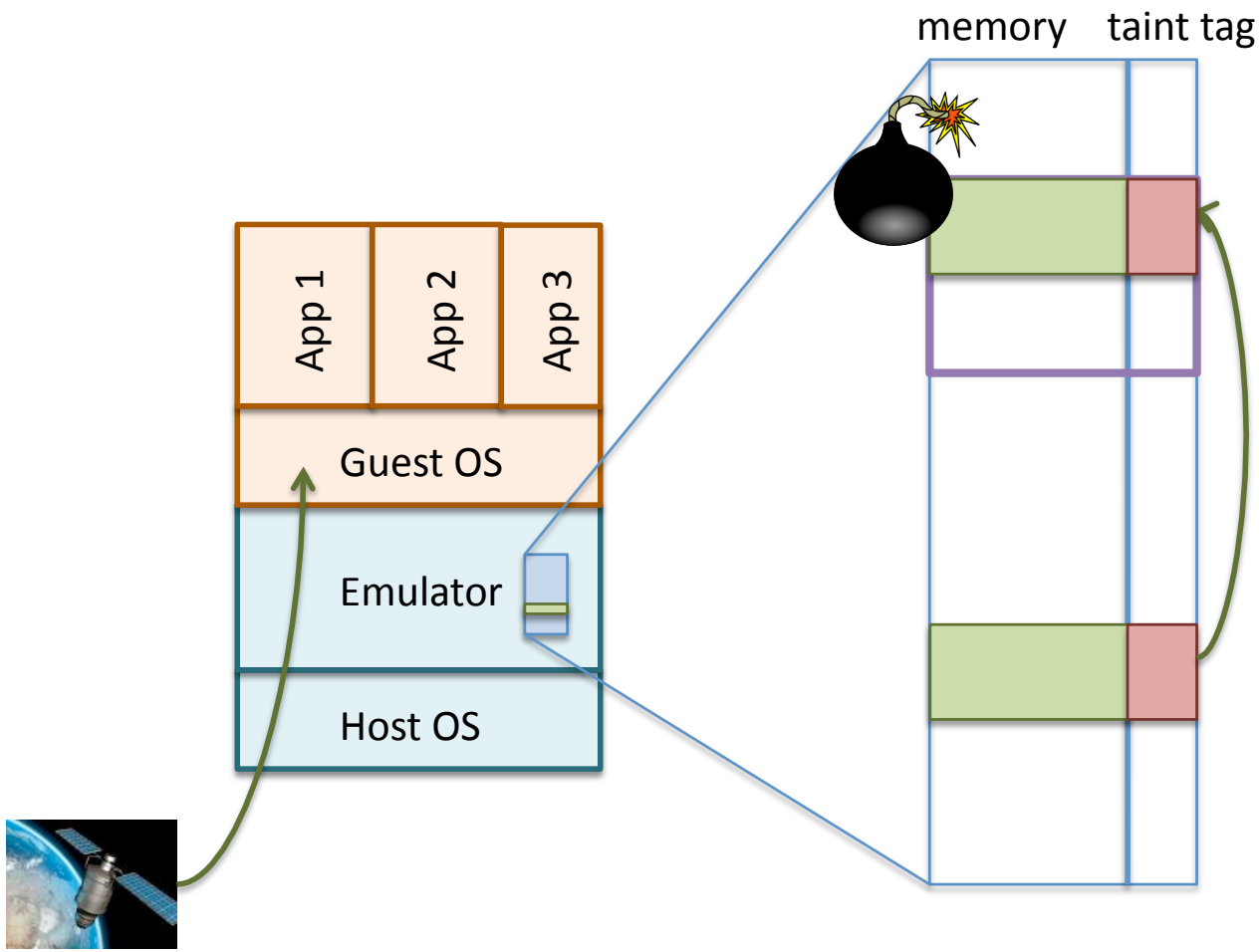
1. Mark network data as tainted.
2. Propagate taint through the OS.

Basic tainting



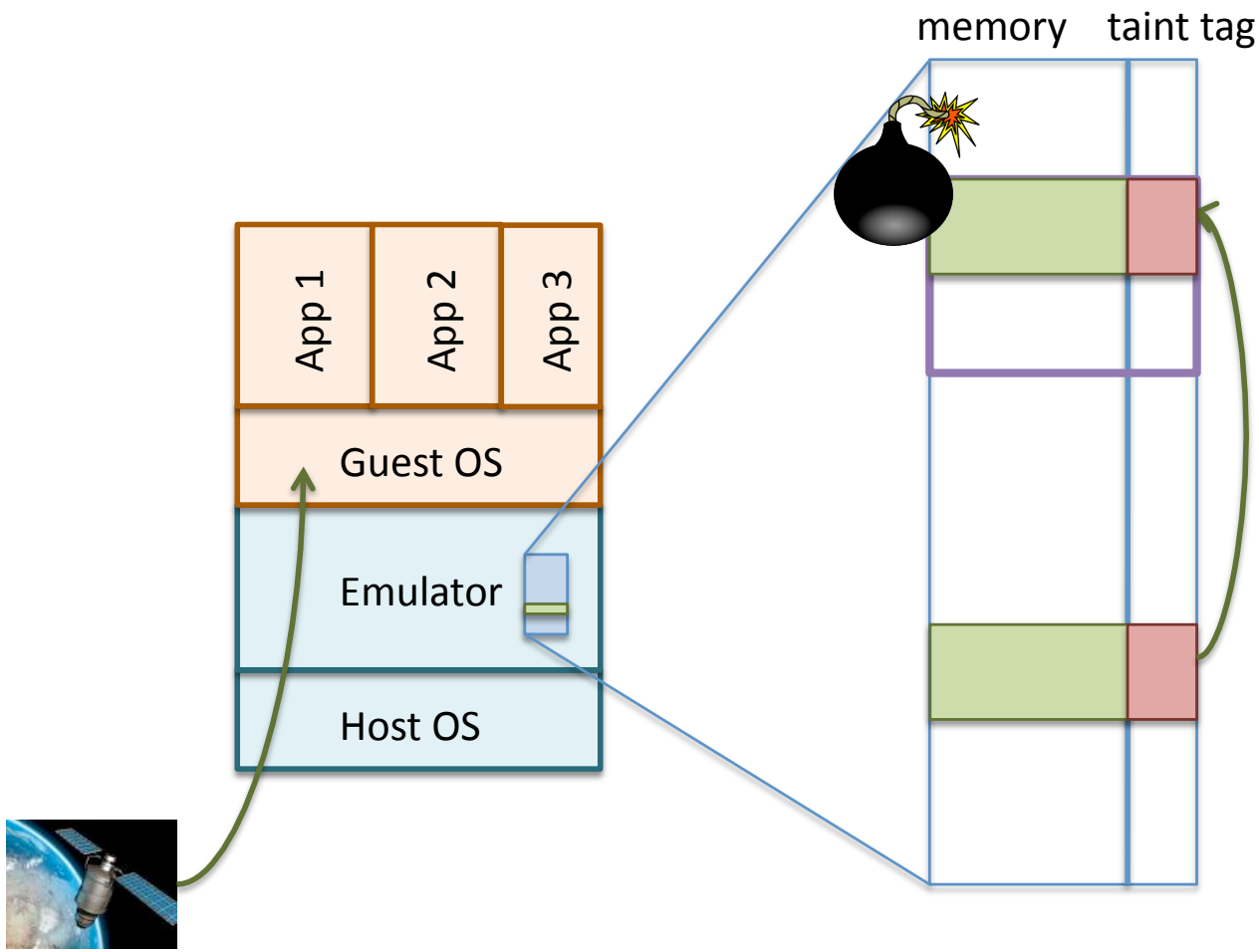
1. Mark network data as tainted.
2. Propagate taint through the OS.
3. Alert for dereferences due to tainted jumps, function calls/ returns.

Basic tainting



1. Mark network data as tainted.
2. Propagate taint through the OS.
3. Alert for dereferences due to tainted jumps, function calls/ returns.

Basic tainting



1. Mark network data as tainted.
2. Propagate taint through the OS.
3. Alert for dereferences due to tainted jumps, function calls/ returns.

Minos, MICRO 2004

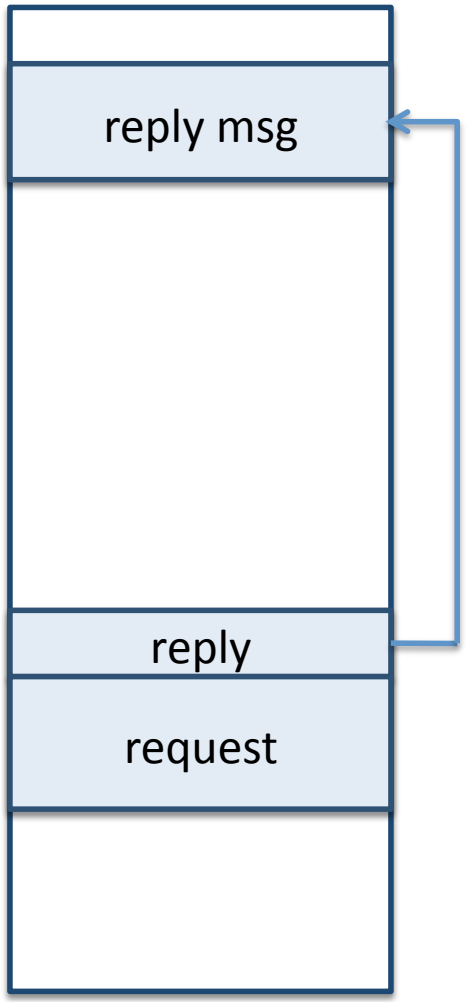
Vigilante, SOSP 2005

Taintcheck, NDSS 2005

Argos, EuroSys 2006

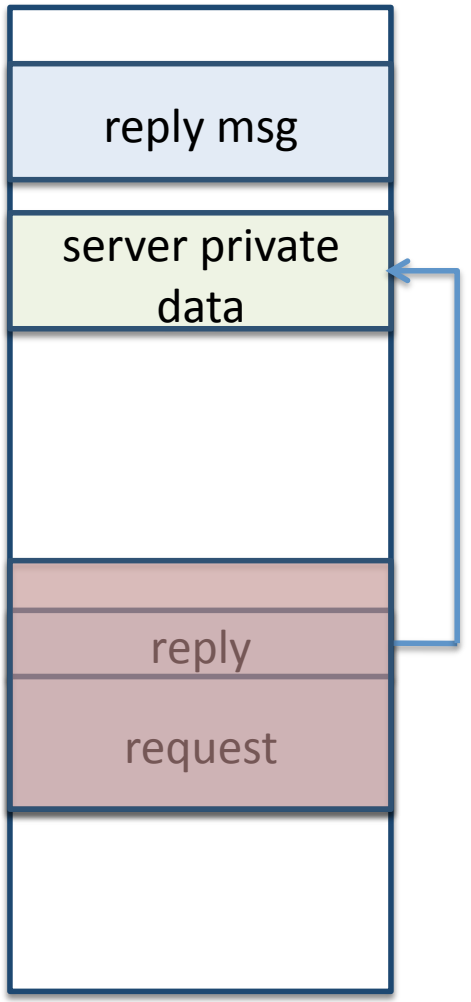
Attacks: (in)effectiveness of basic tainting

```
void serve(int fd)
{
  char *reply = ...;
  char request[64];

  read(fd, request, 128);
  srv_send(fd, reply, 1024);
}
```



Attacks: (in)effectiveness of basic tainting

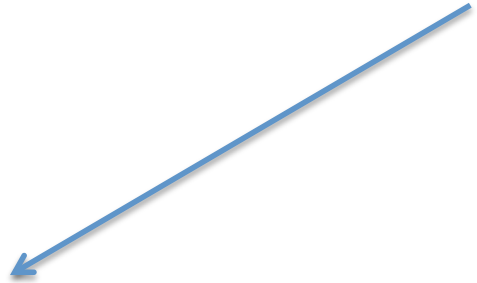
```
void serve(int fd)
{
  char *reply = ...;
  char request[64];

  read(fd, request, 128);
  srv_send(fd, reply, 1024);
}
```



Pointer tainting

1. Mark network data as tainted.

2. Propagate taint through the OS.



- Attacks

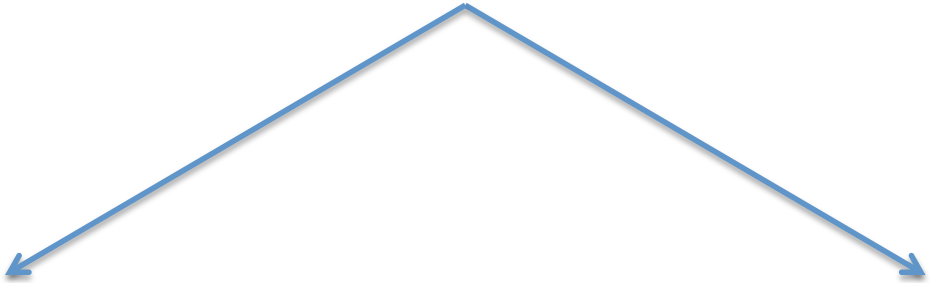
3. Alert for dereferences due to tainted jumps, function calls/returns.

+ If p is tainted, raise an alert on any dereference of p

Pointer tainting

1. Mark network data as tainted.

2. Propagate taint through the OS.



- Attacks

  3. Alert for dereferences due to tainted jumps, function calls/returns.

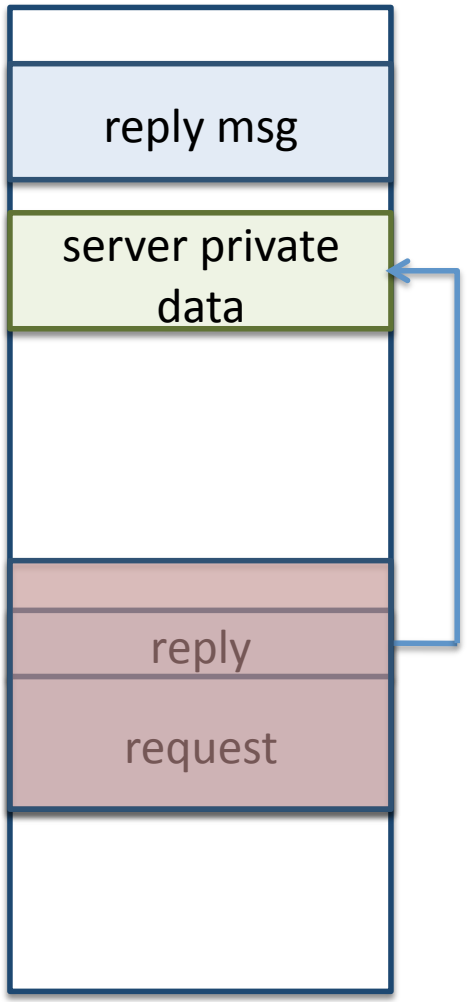
  + If p is tainted, raise an alert on any dereference of p

- Keylogger detection

  + If p is tainted, any dereference of p taints the destination

Attacks: effectiveness of pointer tainting

```
void serve(int fd)
{
  char *reply = ...;
  char request[64];

  read(fd, request, 128);
  srv_send(fd, reply, 1024);
}
```



Pointer tainting: FPs likely

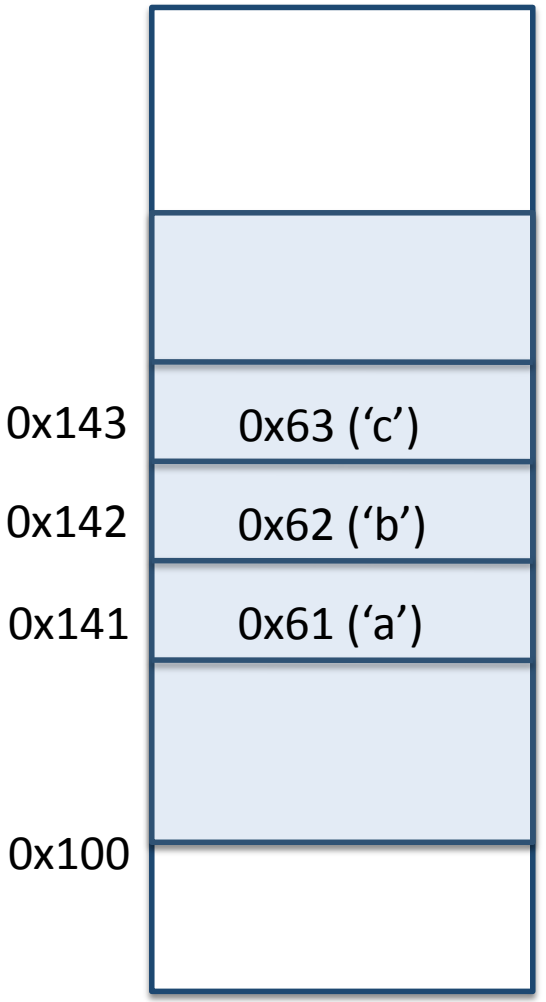
```
void serve(int fd)
{
  char *reply;
  char request;

  read(fd, request, 1);

  srv_send(fd, reply, 1);
}
```

Pointer tainting: FPs likely

```
void serve(int fd)
{
  char *reply;
  char request;

  read(fd, request, 1);
  reply = to_lower[request];
  srv_send(fd, reply, 1);
}
```

Pointer tainting: FPs likely

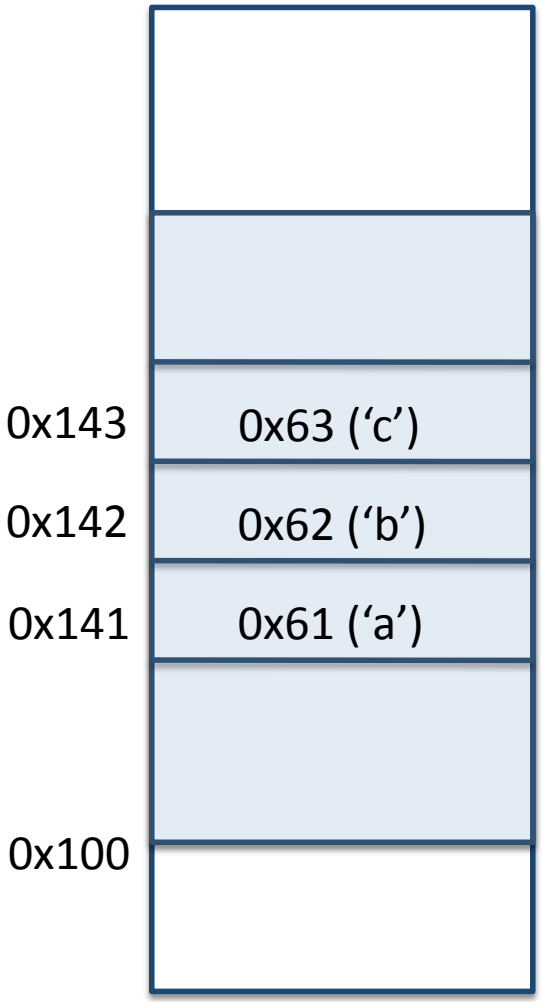
```
void serve(int fd)
{
  char *reply;
  char request;

  read(fd, request, 1);
  reply = to_lower[request];
  srv_send(fd, reply, 1);
}
```



Pointer tainting: FPs likely

```
void serve(int fd)
{
  char *reply;
  char request;

  read(fd, request, 1);
  reply = to_lower[request];
  srv_send(fd, reply, 1);
}
```

request = 0x41 ‘A’



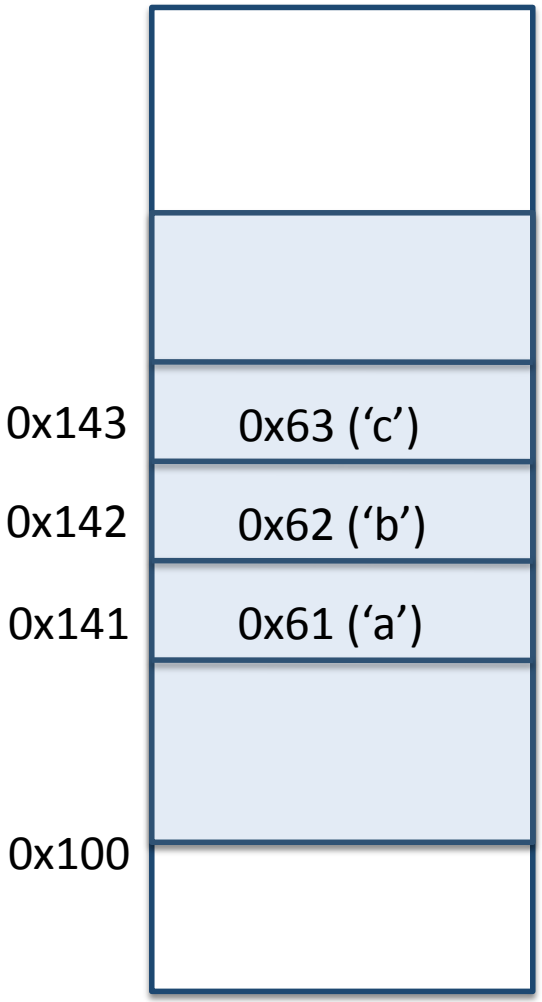
Pointer tainting: FPs likely

```
void serve(int fd)
{
  char *reply;
  char request;

  read(fd, request, 1);
  reply = to_lower[request];
  srv_send(fd, reply, 1);
}
```

request = 0x41 'A'

addr = 0x100 + request

reply = *addr



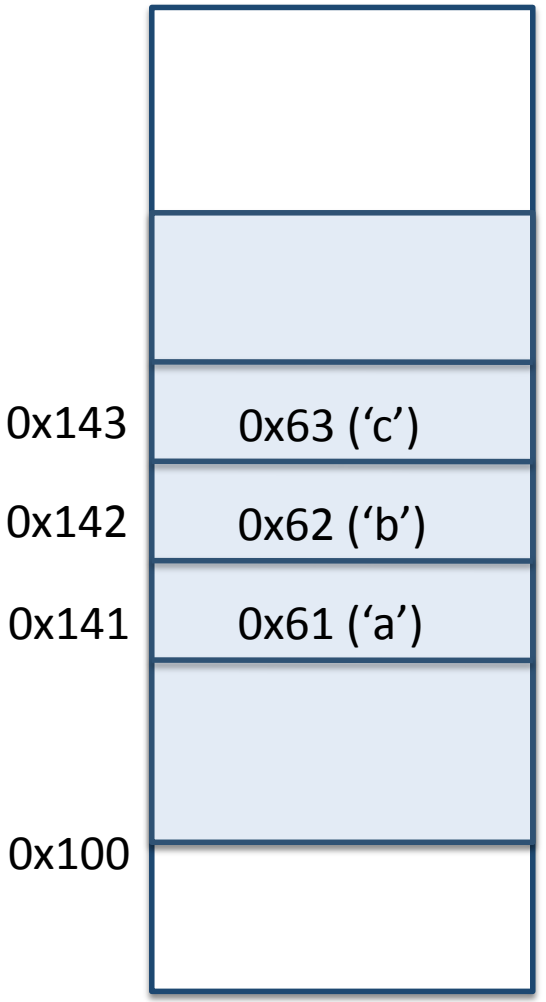
Pointer tainting: FPs likely

```
void serve(int fd)
{
  char *reply;
  char request;

  read(fd, request, 1);
  reply = to_lower[request];
  srv_send(fd, reply, 1);
}
```

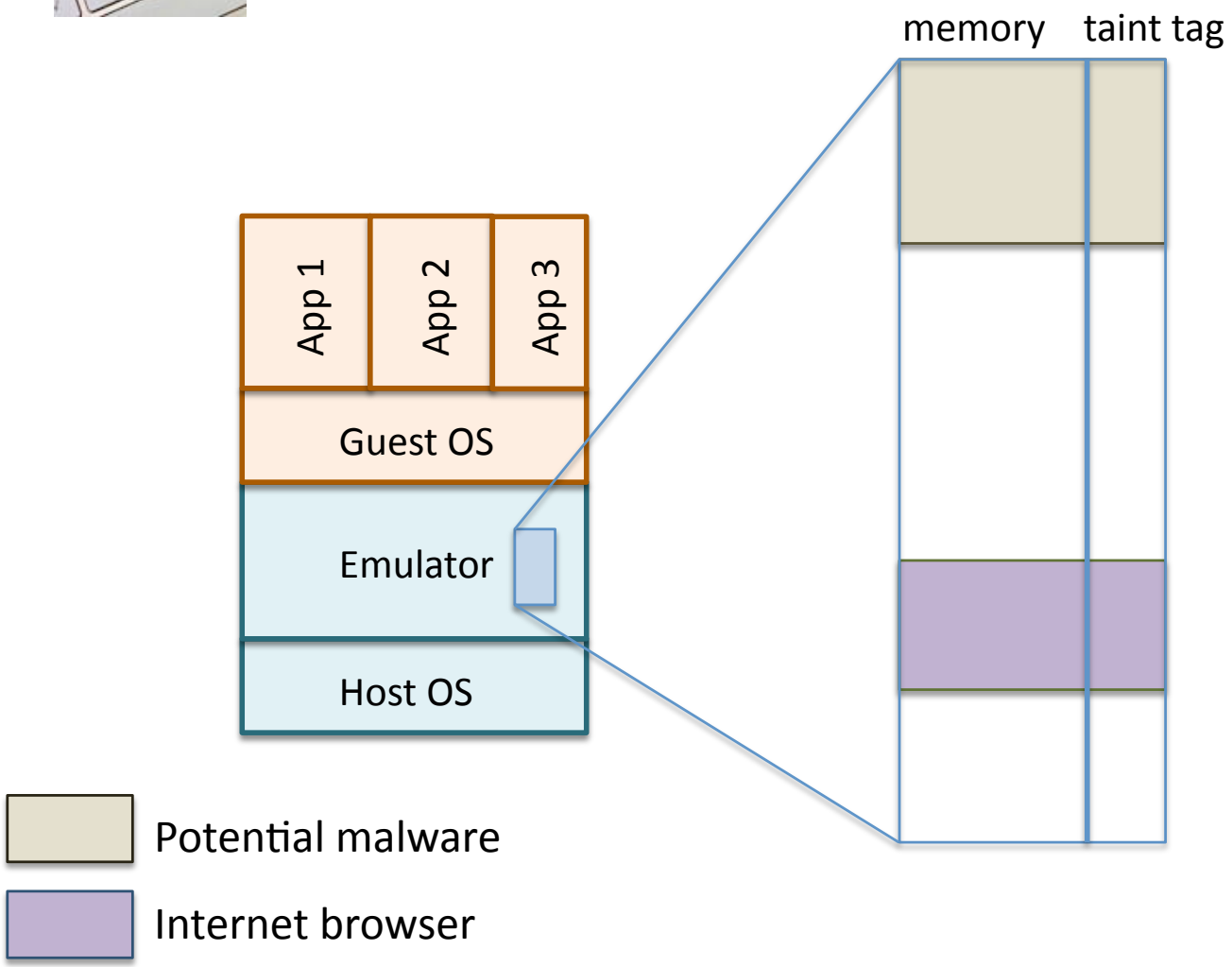
request = 0x41 'A'

addr = 0x100 + request

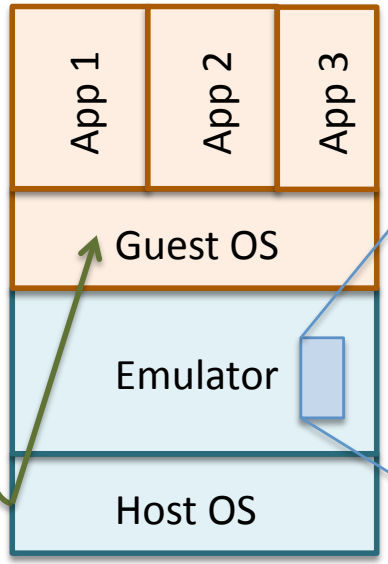
reply = *addr

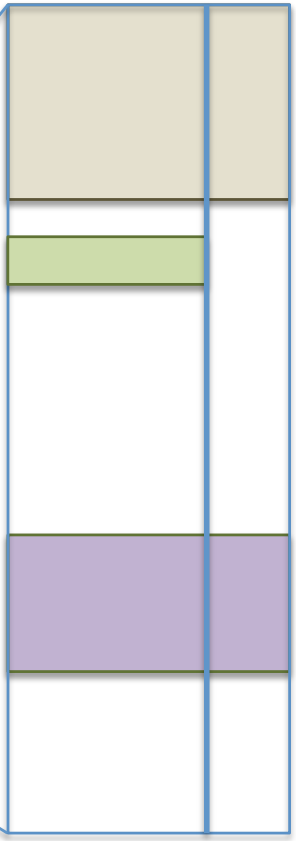


Keylogger detection



Keylogger detection

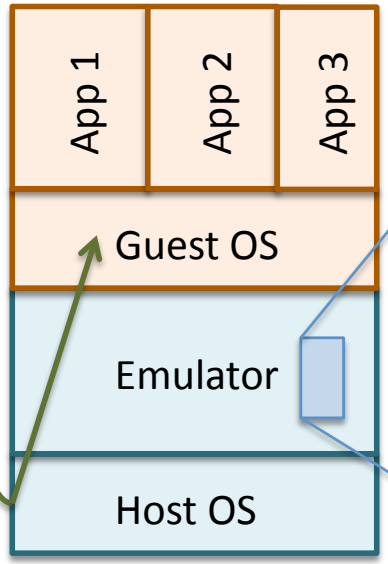


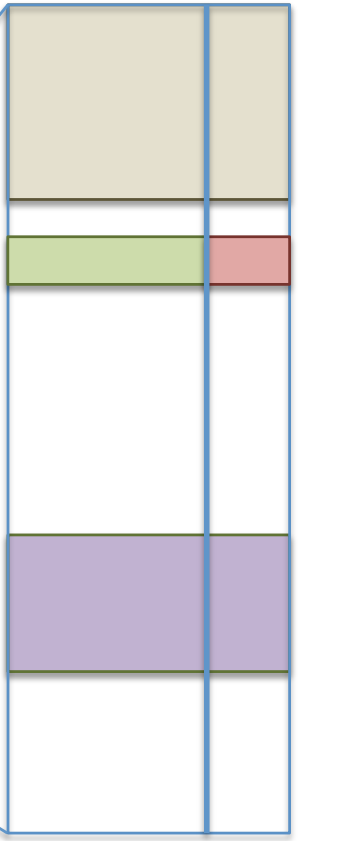


Potential malware

Internet browser

Keylogger detection

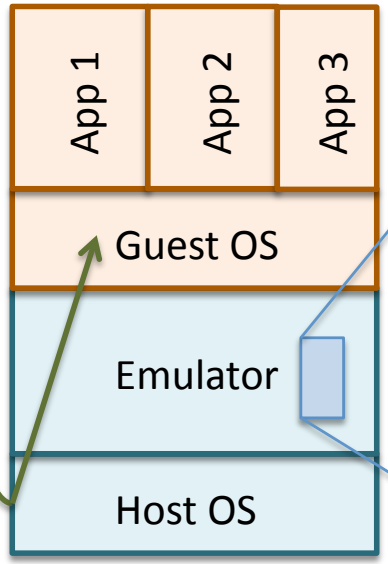


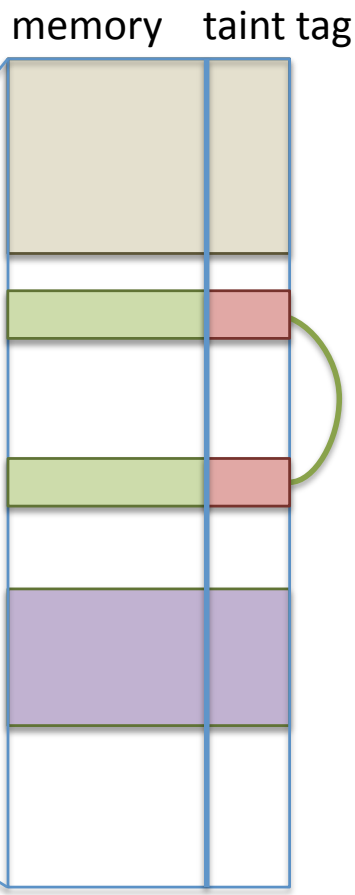


Potential malware

Internet browser

Keylogger detection

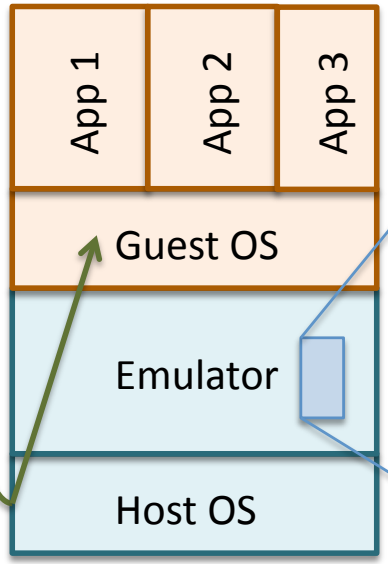


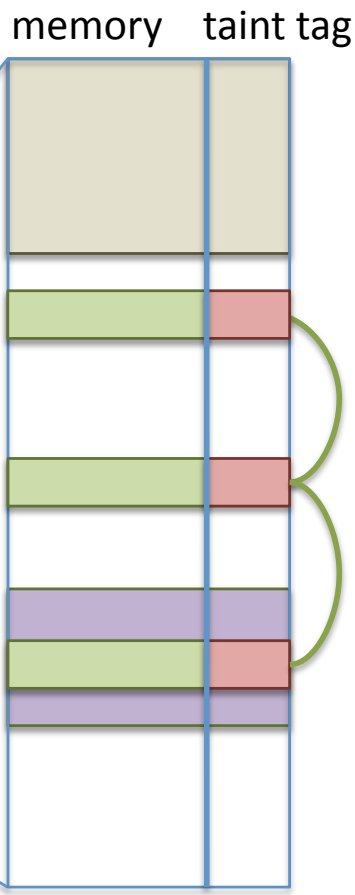


Potential malware

Internet browser

Keylogger detection

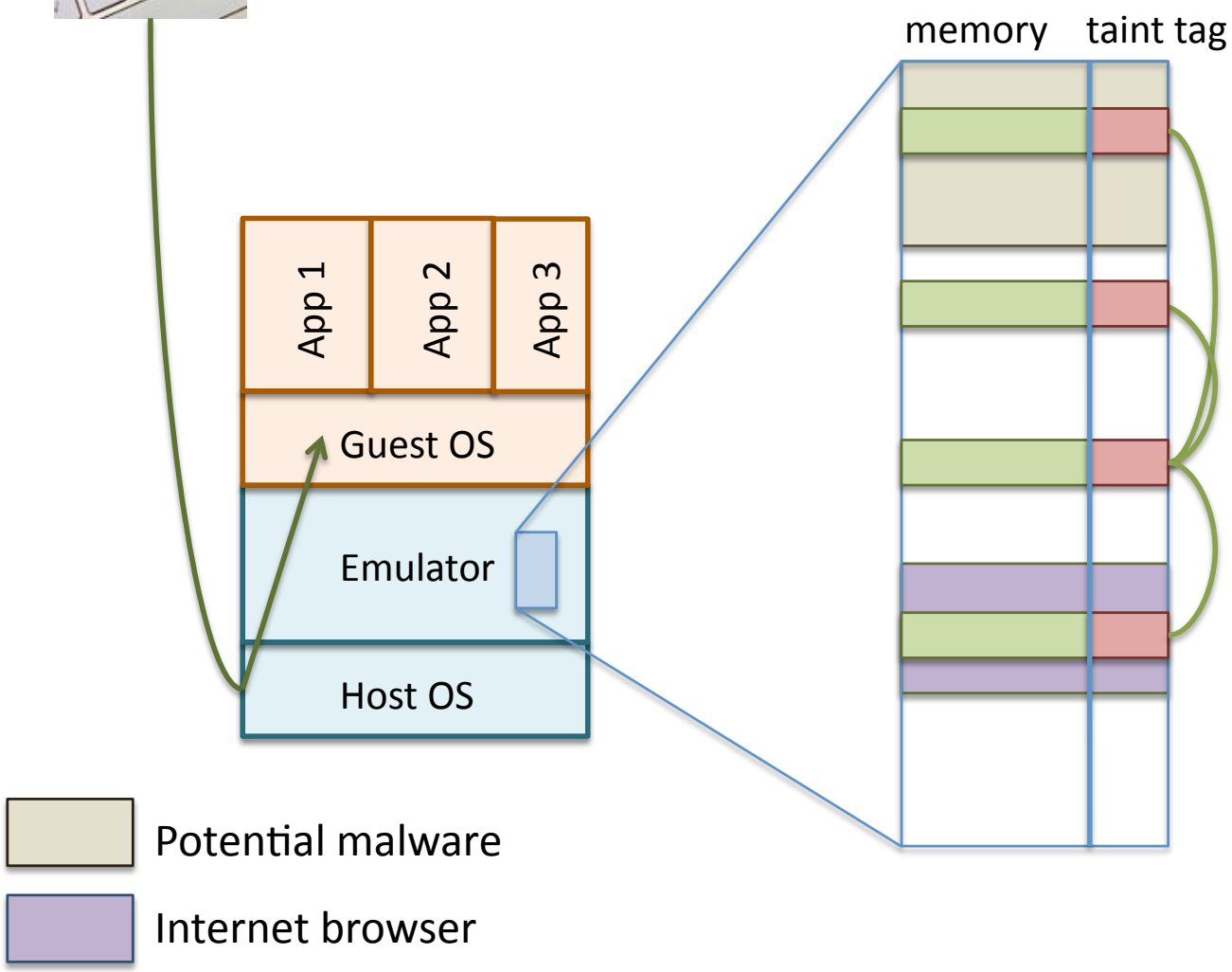




Potential malware

Internet browser

Keylogger detection

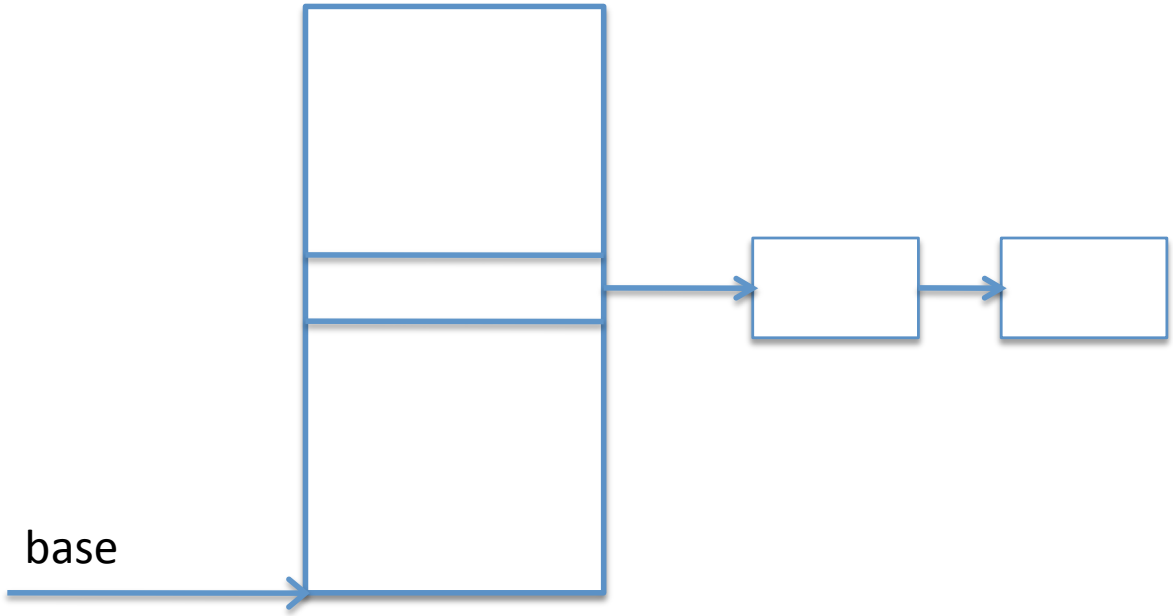


Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
   get_list_head(filename);
```

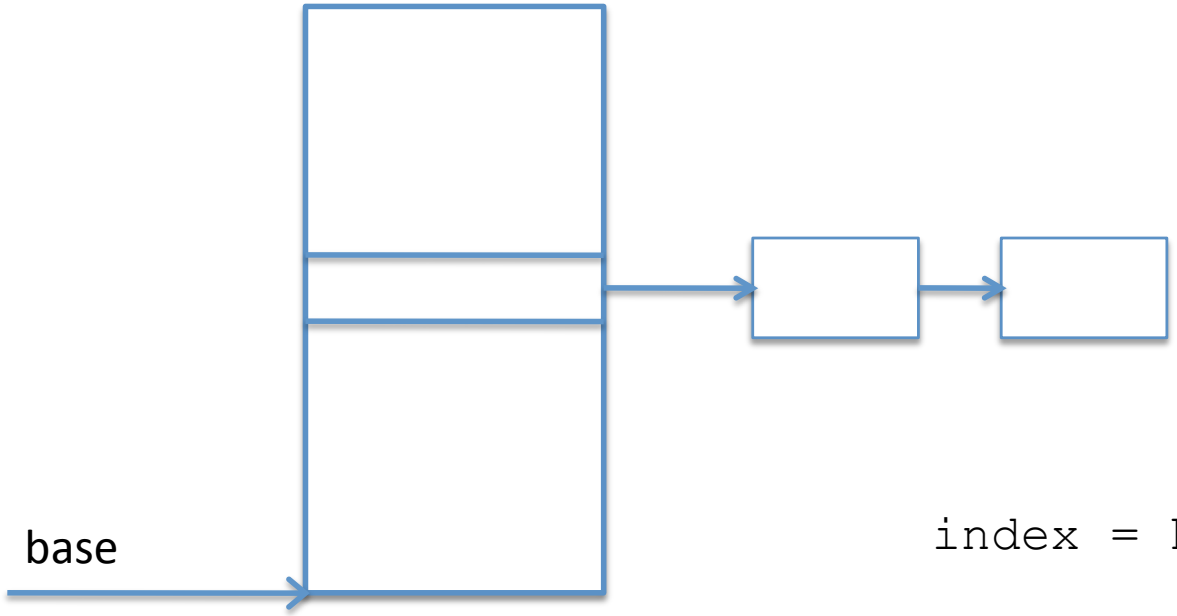
Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
   get_list_head(filename);
```



Keylogger detection: FPs likely (again)

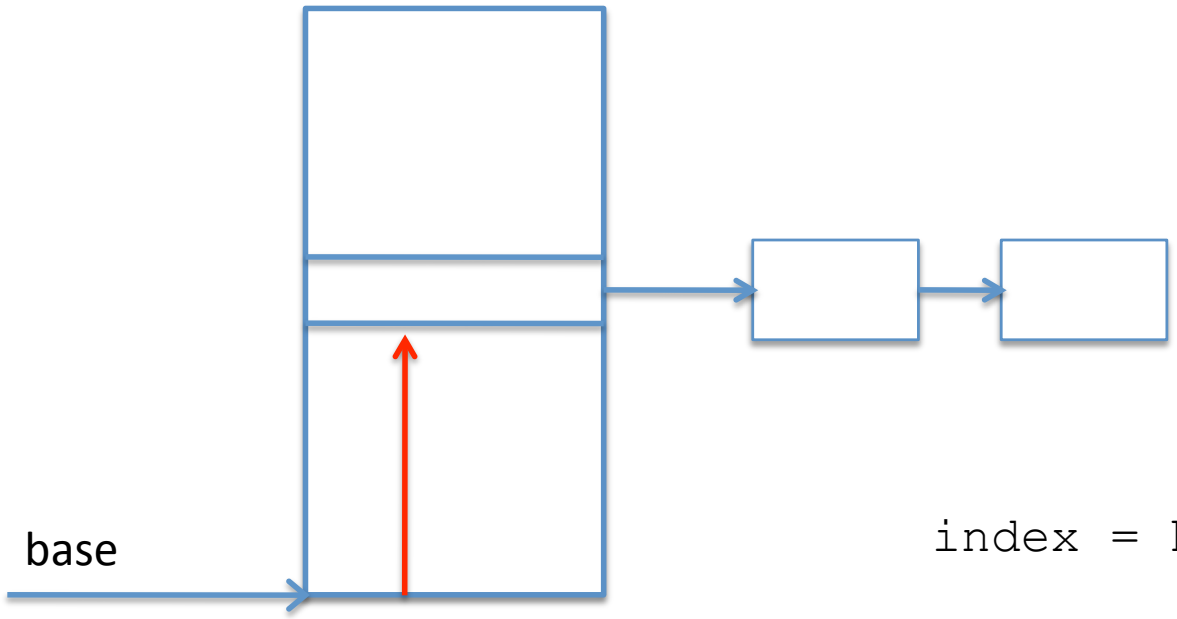
```
struct hlist_head *head =
   get_list_head(filename);
```




```
index = hash(filename)
```

Keylogger detection: FPs likely (again)

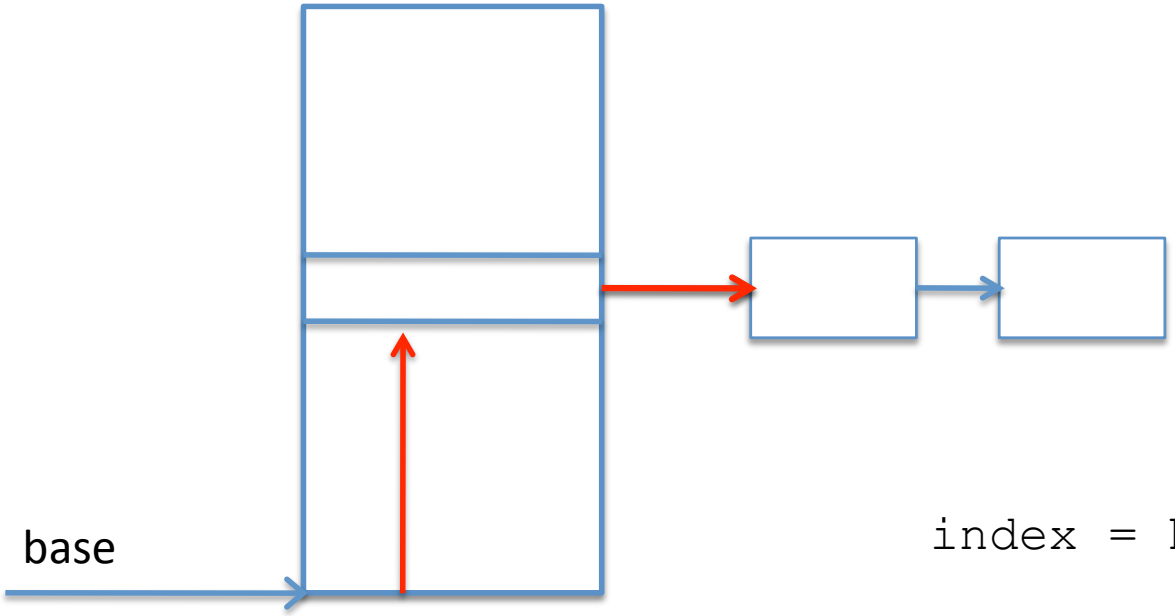
```
struct hlist_head *head =
   get_list_head(filename);
```




```
index = hash(filename)
```

Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
   get_list_head(filename);
```



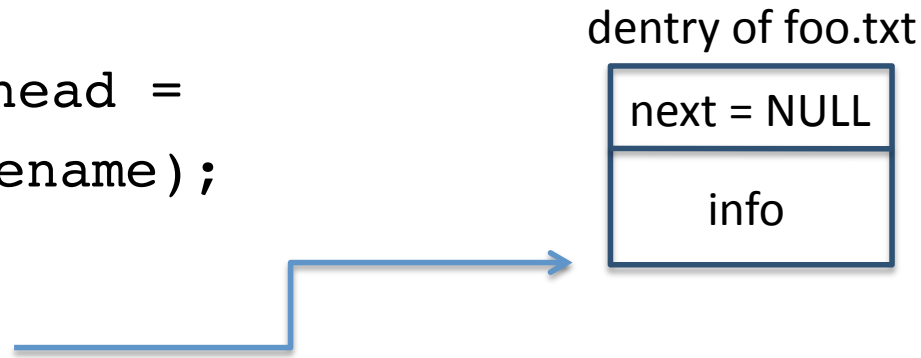

```
index = hash(filename)
```

Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
  get_list_head(filename);

struct dentry
  *dentry = head->first;
```

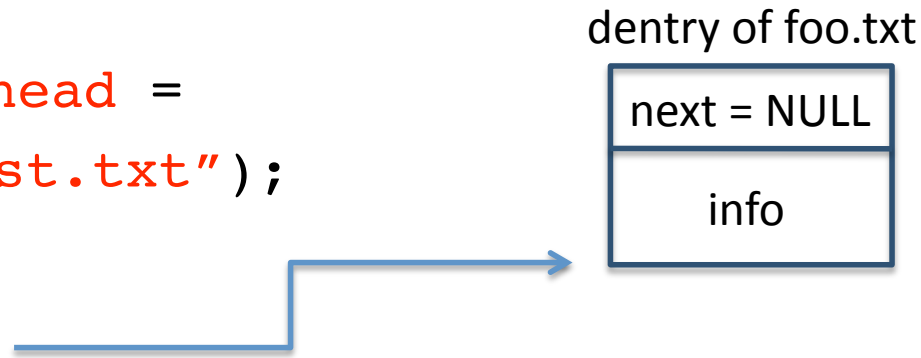
Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
   get_list_head(filename);

struct dentry
   *dentry = head->first;
```



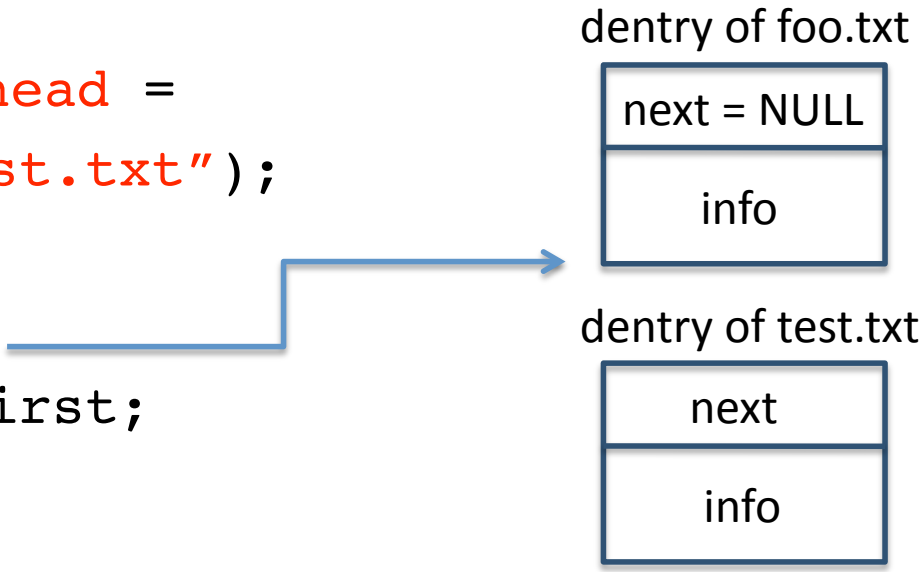
Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
   get_list_head("test.txt");

struct dentry
   *dentry = head->first;
```



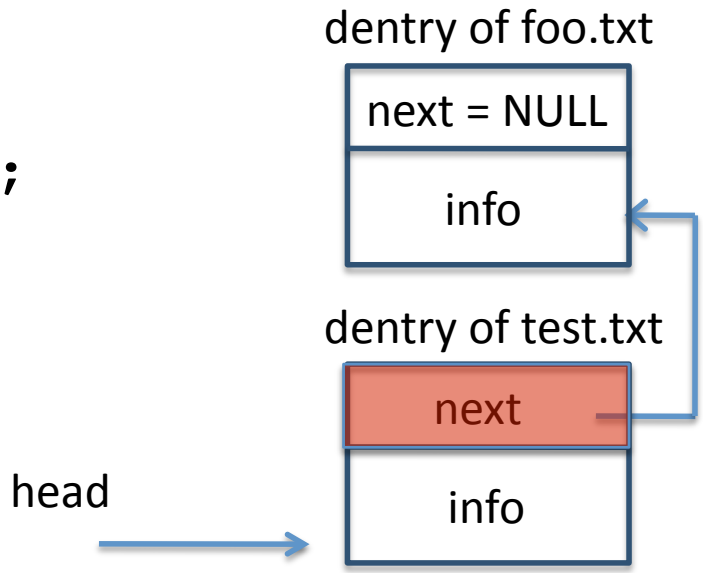
Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
   get_list_head("test.txt");

struct dentry
   *dentry = head->first;
```



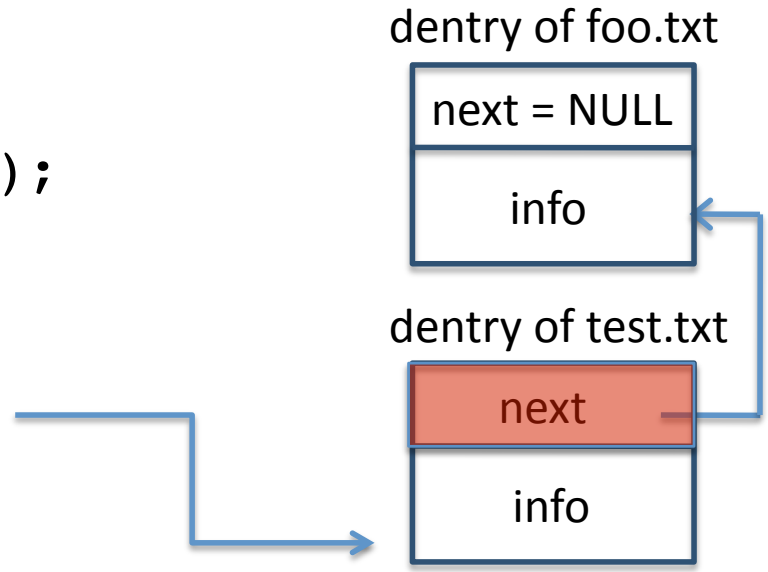
Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
   get_list_head("test.txt");

struct dentry
   *dentry = head->first;
```



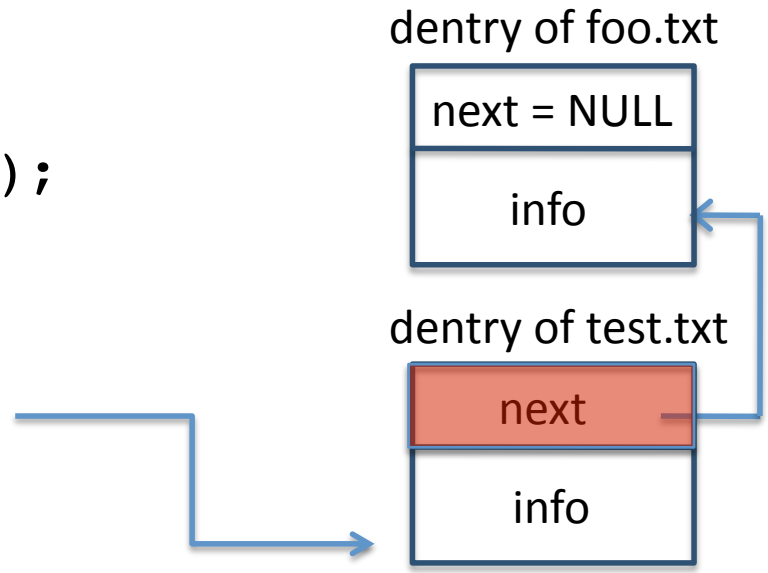
Keylogger detection: FPs likely (again)

```
struct hlist_head *head =
   get_list_head("foo.txt");

struct dentry
   *dentry = head->first;
```



Keylogger detection: FPs likely (again)

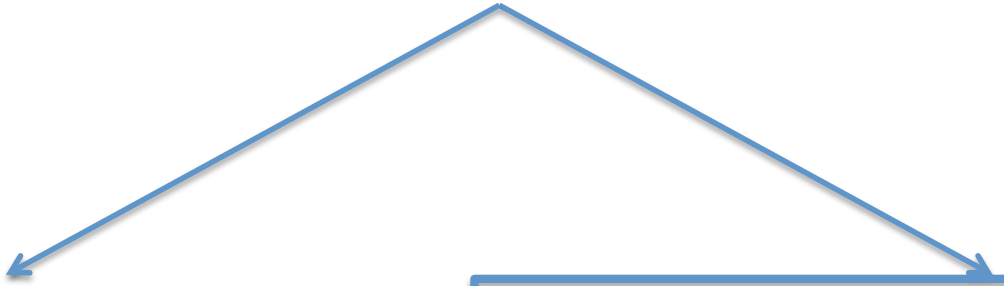
```
struct hlist_head *head =
   get_list_head("foo.txt");

struct dentry
   *dentry = head->first;

dentry = dentry->next;
return info = dentry->info;
```



Pointer tainting

1. Mark network data as tainted.

2. Propagate taint through the OS.



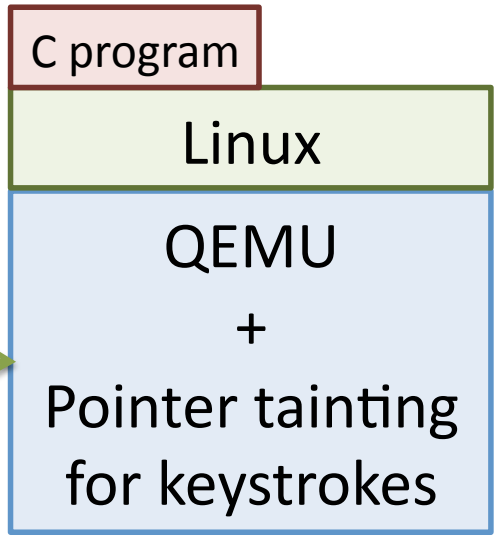
- Attacks

  3. Alert for dereferences due to tainted jumps, function calls/returns.

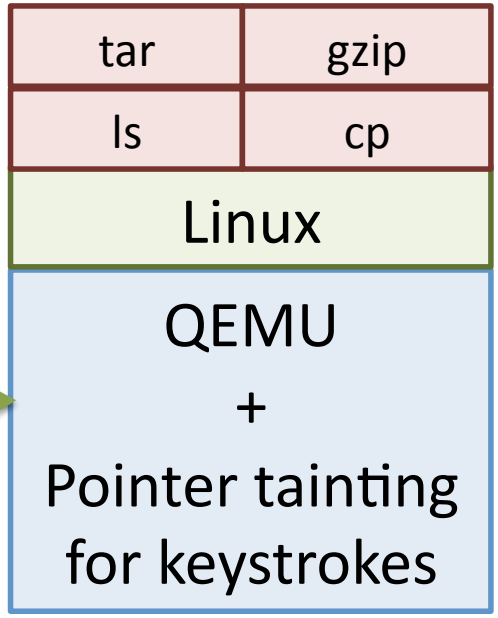
  – If p is tainted, raise an alert on any dereference of p

- Keylogger detection
  – If p is tainted, any dereference of p taints the destination

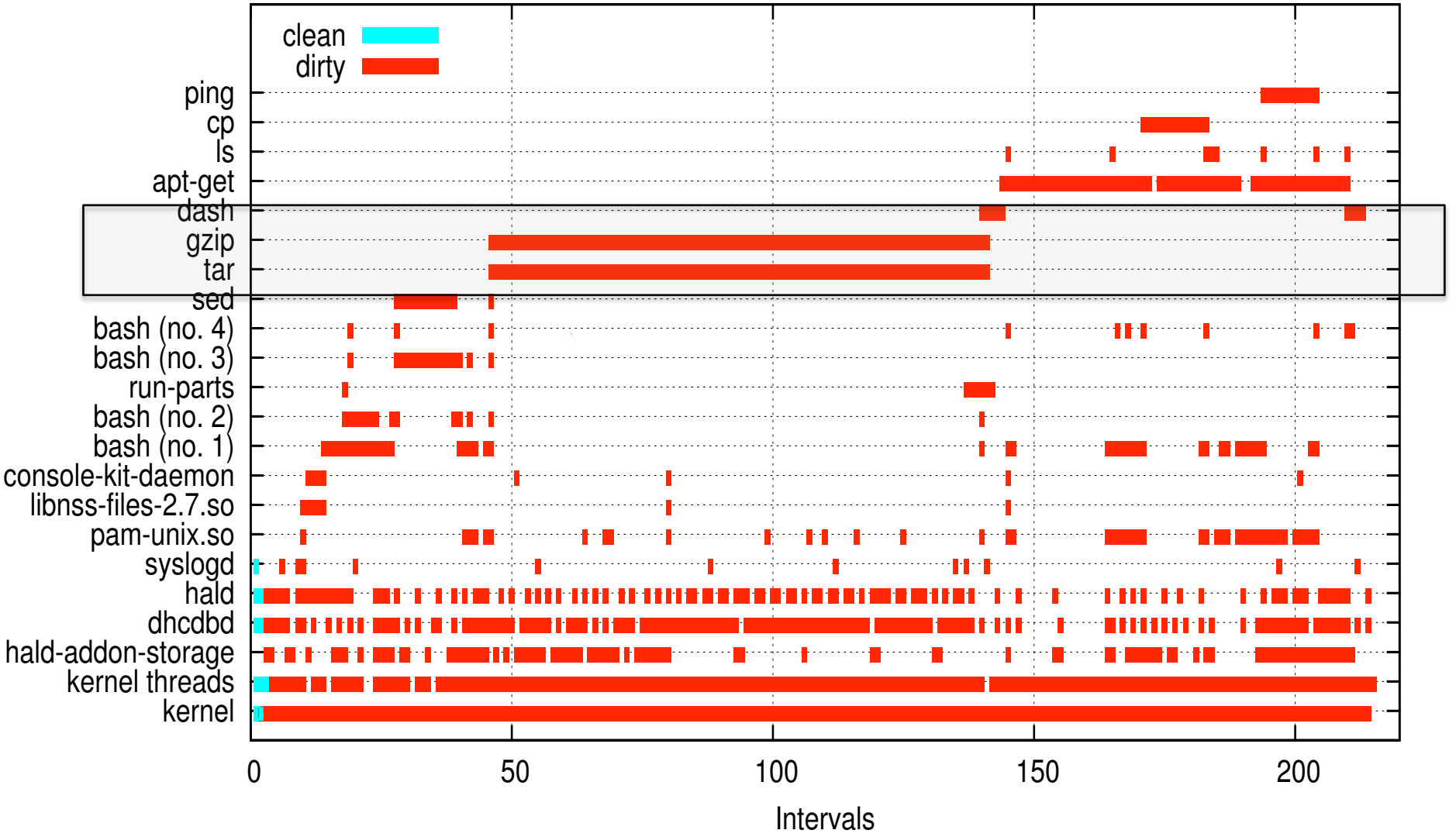
Experiment



Experiment



Keyloggers… false positives!



Containment

- White and black lists
- ESP/EBP protection
- Landmarking
- …

Landmarking - motivation

Should NOT be tainted

```
struct dentry dentry =
   prev_dentry->next;
```

Should be tainted

```
val = transl_table[index]
```

Landmarking - motivation

Should NOT be tainted

```
struct dentry dentry =
   prev_dentry->next;
```

affect tainted address with a **clean** value

Should be tainted

```
val = transl_table[index]
```

Landmarking - motivation

Should NOT be tainted

```
struct dentry dentry =
   prev_dentry->next;
```

affect tainted address with a **clean** value

```
B = prev_dentry + offset

dentry = *B
```

Should be tainted

```
val = transl_table[index]
```

Landmarking - motivation

Should NOT be tainted

```
struct dentry dentry =
   prev_dentry->next;
```

affect tainted address with a **clean** value

Should be tainted

```
val = transl_table[index]
```

affect address with a **tainted** value

```
B = prev_dentry + offset

dentry = *B
```

Landmarking - motivation

Should NOT be tainted

```
struct dentry dentry =
   prev_dentry->next;
```

affect tainted address with a **clean** value

```
B = prev_dentry + offset

dentry = *B
```

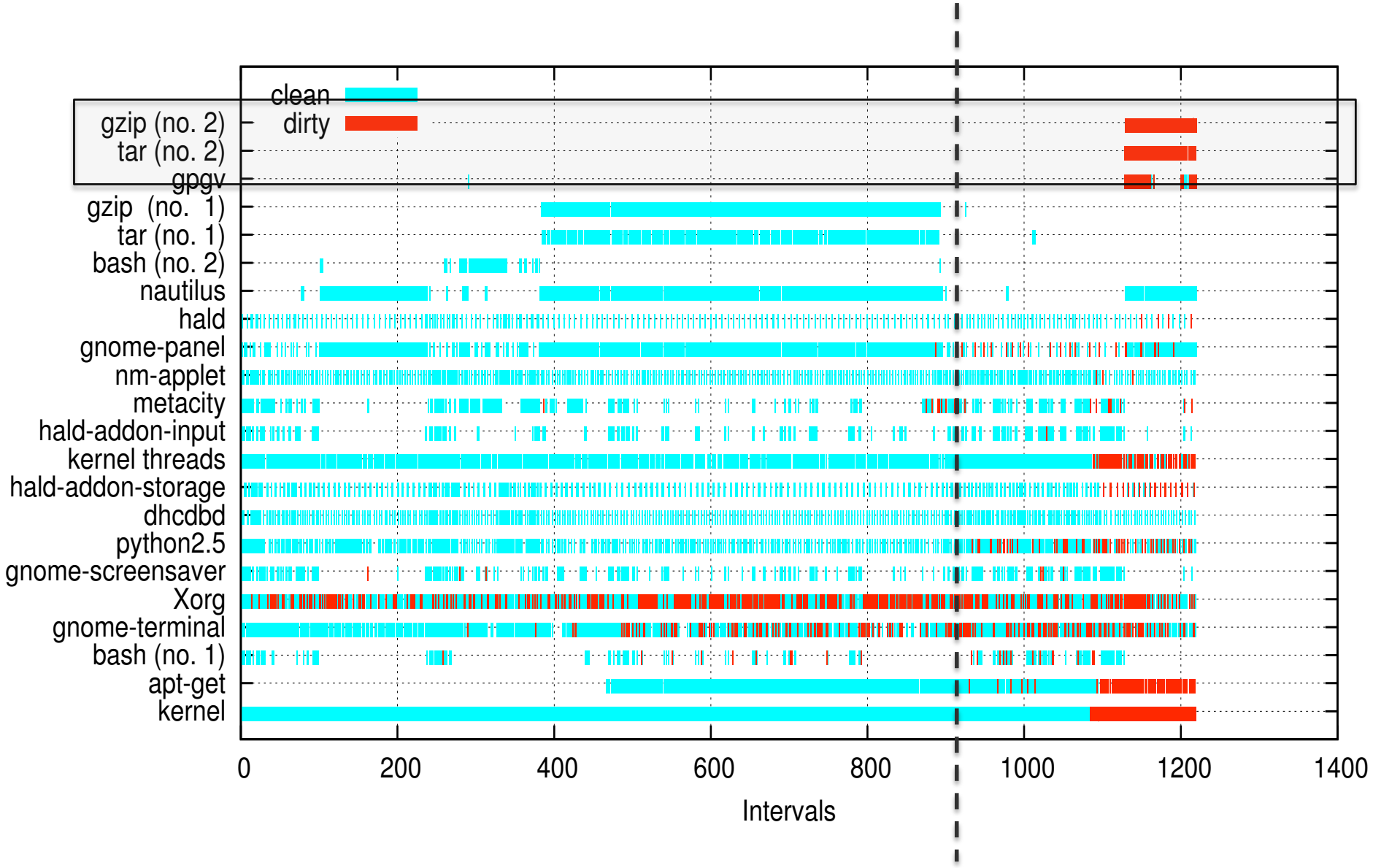
Should be tainted

```
val = transl_table[index]
```

affect address with a **tainted** value

```
A = address of an array
B = A + index*4

val = *B
```

Full containment - results



How bad are things?

Should NOT be tainted

```
struct hlist_head *head =
  d_hash(parent, hash);
```

Should be tainted

```
attributes =
  transl_table[kbd_data];
```

1. `addr`: combine clean pointer with a tainted index

```
struct dentry *dentry =
  head->first;
```

```
lower_case =
  attributes->lower;
```

2 `new_addr`: modify the resulting pointer with a constant

Conclusions

- We have analyzed pointer tainting
  - A popular technique for detecting memory corruption attacks and keyloggers
- Not suited for detecting privacy-breaching malware, like keyloggers
  - False positives hard to avoid
- Could be applied to detect memory corruption attacks
  - Not suitable for x86 and Windows

Backup slides

Pollution due to tainted ESP/EBP

- If ESP/EBP get tainted, taint spreads instantly
  - `mov eax, dword ptr [ebp + 08h]`
  - `pop eax`
- How ESP/EBP can become tainted?
  - Linux kernel has numerous places where it can happen,
    - E.g., a common operation like opening a file ends up tainting EBP,
    - Details in the paper

Pollution due to pointer arithmetic

Should NOT be tainted

```
struct fd {
   HANDLER handler;
   STRING filename;
   struct fd *next;
};

A = address of filename
B = A – 0x0004

h1  = *(B + 0x0000)
fd2 = *(B + 0x0020)
```

Should be tainted

```
A = address of an array
i = index to be accessed
B = A + i*4

Translated value:
val = *B
```

Pollution due to pointer arithmetic

Should NOT be tainted

```
struct fd {
   HANDLER handler;
   STRING filename;
   struct fd *next;
};
```

A = address of filename

B = **A** – 0x0004

h1 = *(**B** + 0x0000)

fd2 = *(**B** + 0x0020)

How to distinguish between these two cases?

Should be tainted

A = address of an array

i = index to be accessed

B = A + **i***4

Translated value:

val = ***B**

Landmarking

```
typedef struct test_t {
   int i;
   struct test_t *next;
} test_t, *ptest_t;
ptest_t table[256] = …;

ptest_t i1 = table[index];          // tainted
   A = (table+index*sizeof(test_t))

ptest_t i2 = i1->next;              // clean
   addr: *(A + offset(next))

int i3 = i1->i;                     // clean
   addr: *(A)
```

Landmarking – why FPs?

- Possible scenarios:
  - Assume eax contains a calculated tainted address
    - It can be copied and altered before dereference
    - Then both values become tainted
  - Addresses calculated directly
    - an array A of struct {int a; int b;}
      - A[index].b: `int b = *((char*)A+8*index+4)`
    - Very simplistic, but the same problem might hold for queues, stacks and hashtables

Landmarking – more problems

- False negatives
  - Translation table containing structures instead of single elements

```
attributes = transl_table[kbd_data];
lower_case = attributes->lower;
```

- Much more problems in the paper