Operating Systems - Advanced

Dynamic Spyware Analysis

Manuel Egele[*], Christopher Kruegel[*], Engin Kirda[*], Heng Yin[‡], and Dawn Song[§]

[*] Secure Systems Lab
Technical University Vienna
{pizzaman,chris,ek}@seclab.tuwien.ac.at

[‡] Carnegie Mellon University and College of William and Mary
hyin@ece.cmu.edu

[§] Carnegie Mellon University
dawnsong@cmu.edu

Spyware

- Installs itself without the user being aware
- Monitors user behavior
- Collects information like:
  - passwords
  - credit card numbers
  - visited sites
- Interferes with the existing software (intentionally or unintentionally)

Browser Helper Objects

- Most Spyware installs as BHO
- A BHO is a library, a plugin for Microsoft Internet Explorer
- A BHO has access to the entire DOM and to all IE events
- Example BHO spyware:
    - activates itself upon detection of a SSL connection, records all keyboard events then sends them to a webserver

Existing Anti-Spyware

- Similar to Anti-Virus products
- Technology based on signatures
    - does not protect against "zero-day" attacks
- Signatures are collected manually
    - expensive, Anti-Spyware vendors analyze hundreds of samples
- Regular updates are necessary
- Simple obfuscation techniques can be employed

The problem

"A distinctive characteristic of spyware is that a spyware component (or process) collects data about user behavior and forwards this information to a third party. Thus, a BHO is classified as spyware when it (i) monitors user behavior (ii) then leaks the gathered data to the attacker."

Scope of solution

- Semi-automated classification of BHOs into benign and malign
- Detailed reporting of the BHO behavior

Approach

- Do a dynamic analysis of the information flow in the browser and the associated BHOs
    - Use “taint analysis”
    - Identify “leaked” data

Benefits

- Detection of stolen data:
  - E.g. URLs, snippet of a Web page, etc
- Detection of how data is transported:
  - E.g.: sent over the net, stored in a file and sent from another process, etc.

Tainting

- Interesting data is marked then tracked throughout the system
  - E.g.: copying a tainted byte A to a memory location B marks B as tainted

Tainting - applied

- Starts by marking as tainted the URLs and the content of web pages
- Continues by tracking the data through the browser code and then BHO code
- If the tainted data gets “stolen” - i.e. is sent to a socket or to a file, the action is recorded and the BHO is marked as spyware

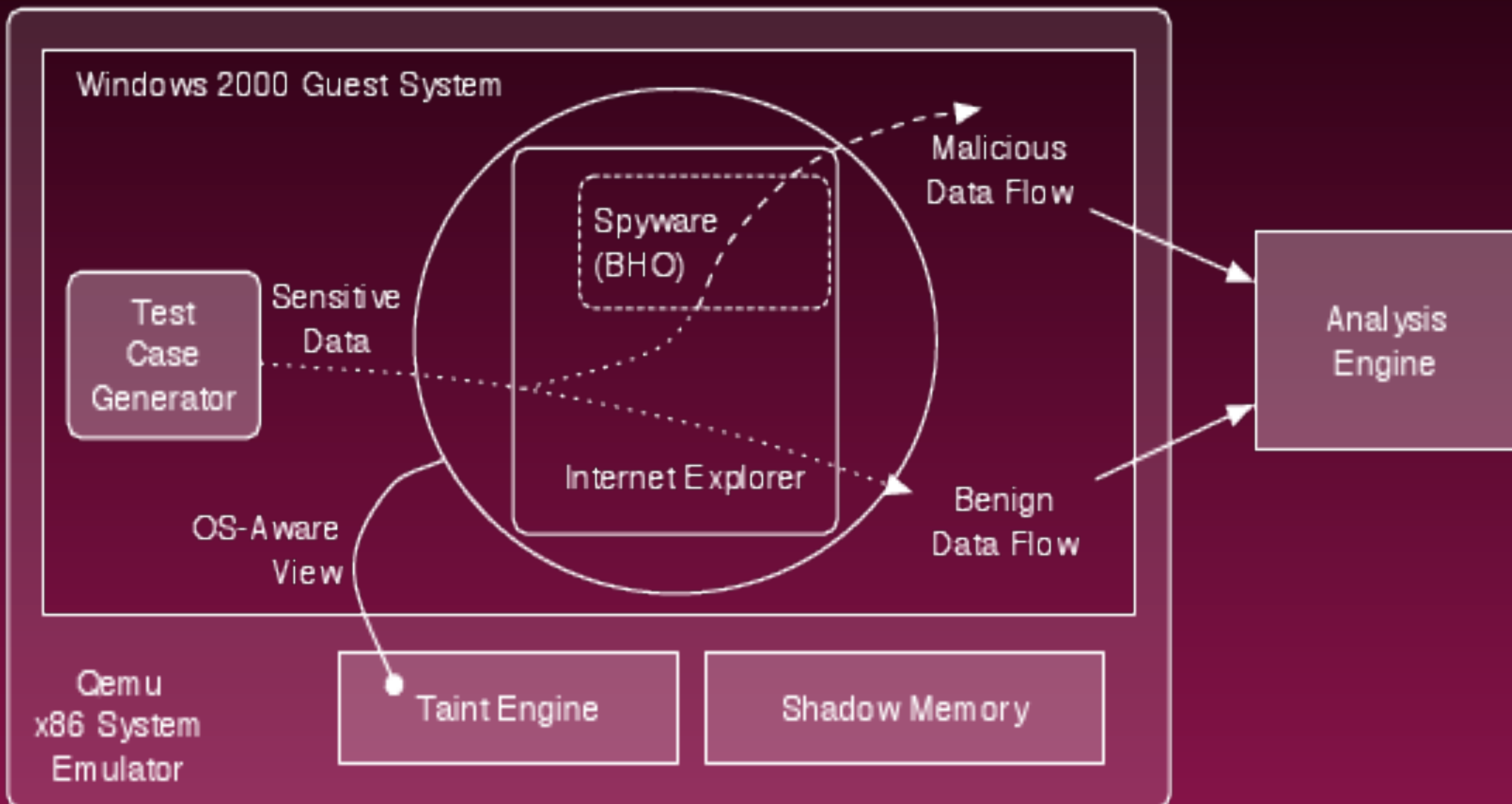
Conclusion

- It is necessary to use **system level** taint analysis.
- Interesting data is stored in **registers** and **physical memory**.
- Data needs to be tracked in kernel-space as well.

Therefore...

- The tainting system needs to know:
  - when an instruction is run in kernel mode
  - when an instruction runs in the context of a certain process
  - moreover, when an instruction runs in the context of a BHO
- We need:
  - “operating system awarness”

System architecture



System architecture (2)

- QEMU/ Windows 2000 / x86 / IE
- Shadow memory – one byte for every byte of physical memory plus the registers
  - a byte is necessary istead of a bit in order to use multiple labels
  - a certain area can be accessed by both IE and the BHO

How to test

- The BHO is installed
- IE is launched - loads the BHO
- Launch the testcase generator - simulating browsing sessions
- Mark URLs and page content
- Track sensitive data using the taint analysis system

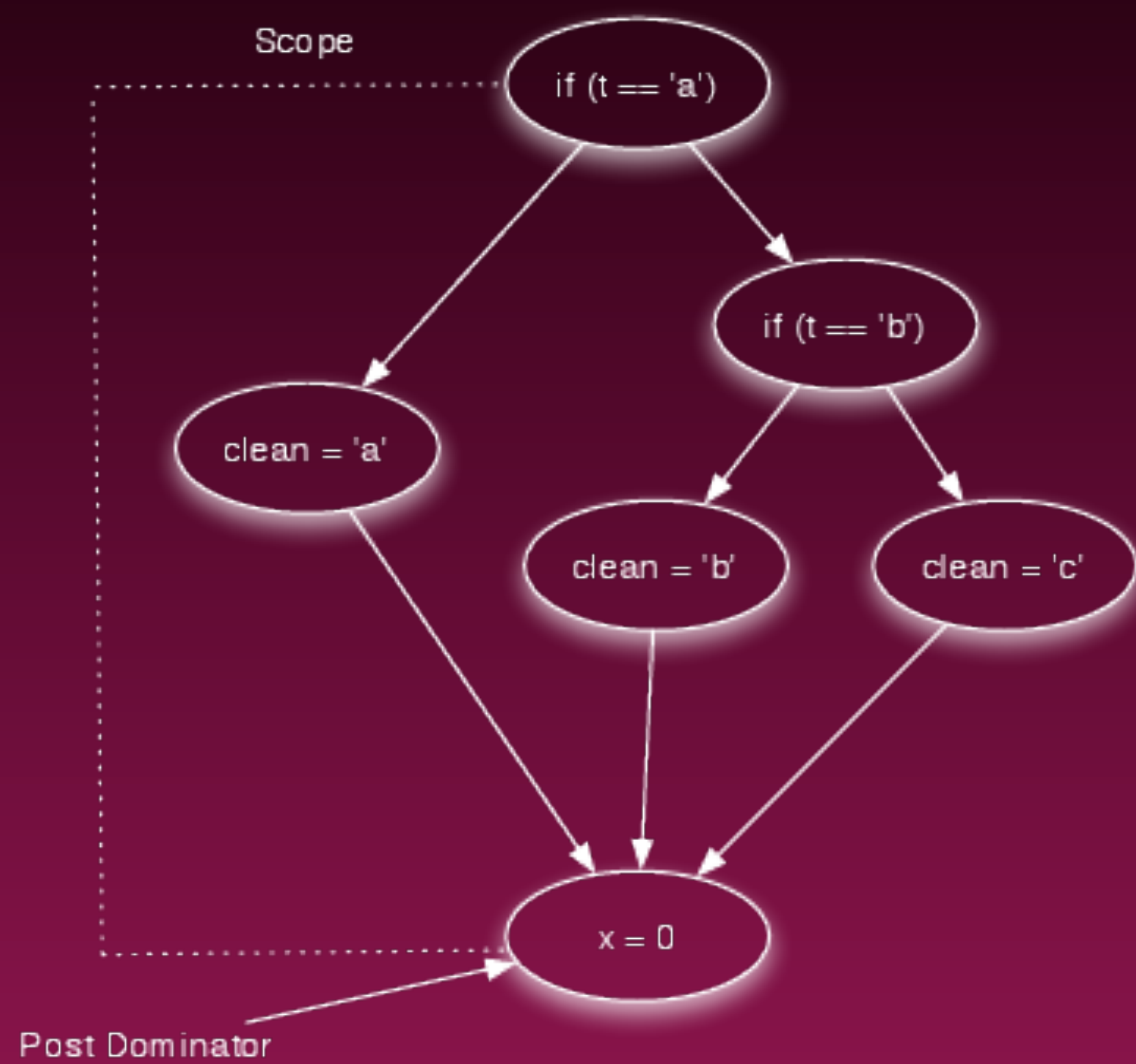
Dynamic taint propagation

- Data dependencies
  - marks all outputs for operations that have one input tainted
  - an entry is considered tainted when an index is tainted
- Not enough!
  - control dependencies need to be investigated

Control dependencies

```
if (t == 'a')
   clean = 'a';
else {
   if (t == 'b')
      clean = 'b';
   else
      clean = 'c';
}

x = 0;
```



Control dependencies (2)

- In the example **t** was propagated as **clean**
- To solve this, we need to identify all instructions associated with a conditional branch and considered as having tainted inputs

How? Static analysis

- Finding the “post-dominator” - the instruction after which we stop
- Build a partial CFG (Control Flow Graph)
  - start at the branching instruction
  - follow all paths until they all intersect (Lengauer-Tarjan)
  - the solution uses a recursive disassembler

Problems

- ret, jmp instructions to unresolved targets
  - assumes the executable is not “self modifiable” because the system detects this behavior and marks the BHO malign
- The CFG can be incomplete
  - more than one post-dominators -> marks the BHO as malign

Untainting

- We need to clear taint status:
  - when an operation with all inputs untainted has the output in a tainted location
  - when constants are propagated into tainted zones
    - e.g. xor %eax, %eax;

Identifying entities

- Qemu offers a hardware level view of the system: registers, physical memory, I/O ports
- We need to identify: processes, user, kernel, BHO

Identifying processes

- We use the CR3 register
    - holds the page table of the current process
    - every process has a unique address space
    - every process has a unique CR3
- If we can map CR3 to processes - we know if the current instruction executes in the context of that process

Finding out CR3 for IE

- Intercepts NtCreateProcess
  - checks that EIP is the NtCreateProcess start address, known by looking into ntoskrnl.exe
  - checks the process name
- Complication: virtual memory
  - Qemu accesses physical memory
  - Solution: manually translate virtual into physical address by using the CR3 page table

Identifying the BHO

- Obvious solution
  - all instructions have the EIP in the text segment of the BHO
  - has a problem
- What if the BHO calls code in another library or from IE itself? Solution:
  - when the control is transferred from the IE to BHO, record the SP value
  - at every modification of the SP, checks the new value to be below the recorded one

How to identify the code segment of the BHO?

- Intercept LdrLoadDLL
  - maps a library, BHO, etc into the IE address space
  - returns the start address upon successful completion
- Segment size
  - stored in EPROCESS
  - the EPROCESS of the current process is mapped at a fixed memory location

Other problems

- Threads
  - mess up the SP based analysis
  - solution: identify the thread switch when returning from kernel into user, by looking at thread_id (in the KTHREAD structure)
- Evasion
  - Injecting malicious code into the IE address space
    - needs to change protection - monitored
  - modifying SP
    - we recognize this and record the new SP

Taint sources

- URL strings in memory
  - Intercept IwebBrowser2::Navigate
  - Mark the argument as tainted
- Web pages
  - intercepts NtDeviceIoControlFile (receive) and marks the buffer as tainted
- Use different labels for the each source type

Taint sinks

- Monitor the interfaces through which the tainted data gets "out" of the process
    - network communication (NtDeviceIoControlFile)
    - file saving (NtWriteFile, NtCreateFile)
    - IPC - SHM

Automating the testing process

- We need a browsing session that's long enough to trigger the BHO
- We need record-playback in order to mimic as closely as possible human interaction with the browser
  - Firefox plugin (recorder)
  - W32 app to control IE (replay)
    - gets a browser handle
    - calls IwebBrowser2::Navigate to load the page
    - uses DOM access to complete the forms

Evaluation

	Spyware	False Negative	Benign	Suspicious	False Positive	Total
Spyware	21	0	-	-	-	21
Benign	-	-	12	1	1	14

Network	File System	Registry	Shared Memory	Total
11	1	3	6	21

Table 2: Different mechanisms used by spyware to leak sensitive data.

Detailed analysis example

- Zango: “ad-supported freeware”
  - zangohoo.dll -> BHO installed with IM client
  - every URL is copied in a shared memory
  - zango.exe reads these URLs and sends them over the net

Questions?

- Spyware
- BHO
- DOM
- Taint analysis
- Static analysis
- QEMU
- CFG
- CR3
- zero-day
- shadow memory
- post-dominator
- EPROCESS