Fingerprinting Intruders

Herbert Bos
Vrije Universiteit Amsterdam

Herbert Bos
VU Amsterdam
It Works!

<table>
<thead>
<tr>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache chunked encoding overflow</td>
</tr>
<tr>
<td>IIS ISAPI .printer host header overflow</td>
</tr>
<tr>
<td>WebDav ntdll.dll overflow</td>
</tr>
<tr>
<td>FrontPage Server Extensions Debug Overflow</td>
</tr>
<tr>
<td>War-FTP overflow</td>
</tr>
<tr>
<td>ASN.1 Library Bitstring Heap Overflow</td>
</tr>
<tr>
<td>Windows Message Queueing Remote Overflow</td>
</tr>
<tr>
<td>RPC DCOM Interface overflow</td>
</tr>
<tr>
<td>LSASS Overflow</td>
</tr>
<tr>
<td>Windows PnP Service Remote Overflow</td>
</tr>
<tr>
<td>nbSMTP remote format string exploit</td>
</tr>
<tr>
<td>NetApi exploit</td>
</tr>
<tr>
<td>WMF exploit</td>
</tr>
</tbody>
</table>
more importantly…

Georgios Portokalidis

Asia Slowinska

Vrije Universiteit Amsterdam
Roadmap

- problem
- Argos
- buffer overflow
- attack detection
- tracking data
- forensics
- signatures
- more apps
- conclude

- vuln. sigs
- Argos overhead
- eudaemon speed
- eudaemon details
- apps

- data
- signatures
- vuln. sigs

- more apps
- conclude

Herbert Bos
VU Amsterdam
fast lane: I understand the context, give me details

Scenic route: first introduce the issues

Introduction
Why?

- Too many vulnerabilities
- New (worm) attacks
- Human intervention too slow
- Current solutions are problematic
  - Time consuming
  - Inaccurate
Why is this important?

• worms are too fast to have humans in the loop
  - how to provide fully automated response
  - response time in minutes (or even seconds)
  - ‘zero-day’ worms

July 19, 2001
spread of CODE RED in 24 hours!

Jan 29, 2003
spread of SLAMMER in 30 minutes!
Current trends

- World domination is out!
Current trends

- Narrow spread is in
Polymorphism is in
Why Is It Difficult?

- It is not your father’s Internet
  - tools were developed mostly for slow networks
- It is a *moving* target
  - way of spreading changes
  - worms modify themselves
  - very little reaction time
  - we have no idea what the next threat looks like
Why is this important?

- (hundreds of) billions of $$ of damage
  - destructive worms
  - denial of service attacks
  - theft (e.g., credit card details)
Imagine that…

- (hundreds of) billions of $$ of damage
  - destructive worms
  - denial of service attacks
  - theft (e.g., credit card details)

- threats range from annoying to dangerous:
  - Worms take out alarm phone numbers
Imagine that…

- (hundreds of) billions of $$ of damage
  - destructive worms
  - denial of service attacks
  - theft (e.g., credit card details)

- threats range from annoying to dangerous:
  - Worms take out alarm phone numbers
  - Worms take out cash machines
Imagine that…

- (hundreds of) billions of $$ of damage
  - destructive worms
  - denial of service attacks
  - theft (e.g., credit card details)

- threats range from annoying to dangerous:
  - worms take out alarm phone numbers
  - Worms take out cash machines
  - Worms take down train signalling
Imagine that…

- (hundreds of) billions of $$ of damage
  - destructive worms
  - denial of service attacks
  - theft (e.g., credit card details)

- threats range from annoying to dangerous:
  - worms take out alarm phone numbers
  - Worms take down train signalling
  - Worms wipe out safety control in nuclear reactor

It all happened…
What are we up against?

what would a *Superworm* look like?

- consider a flash worm: spreads fast
- fast is:
  (a) 1 second
  (b) 30 seconds
  (c) 5 minutes
  (d) 10 minutes
  (e) 30 minutes

Not to mention Botnets…
What are we up against?

"what would a Superworm look like?"

– consider a flash worm: spreads fast

– fast is:
  (a) 1 second
  (b) 30 seconds
  (c) 5 minutes
  (d) 10 minutes
  (e) 30 minutes

All is lost…

All is lost…
No!
Goals
- Platform for next generation honeypots
- Protect entire OS
- Detect most common attack vectors
- Accuracy
Current Solutions

- Honeypots
- Payload based anomaly detection
- Automatic updates & firewalls
Honeypots

- Passive
- Good for IP scanning based attacks
- Insufficient for network specific malware today
- How about future hit-list worms?
Payload Based Anomaly Detection

- Deployed at the network backbone
- Detects unknown attacks
- Prone to false positives
- Can’t cope with increasing net speeds
- What about encryption?
Automatic Updates & Firewalls

- Applied at the host
- Address vulnerabilities at the source
- It is not used widely due to user ignorance
- Illegal software is not supported
- Usability issues with firewalls
Argos Overview

- Applications
- Guest OS
- Argos Emulator
- Host OS

Herbert Bos
VU Amsterdam
Argos Overview

- Forensics
- Applications
- Argos Emulator
- Guest OS
- Snitch
- Host OS
- Signature
- Post-Processing Sub-system

Herbert Bos

VU Amsterdam
no need to explain: I already know or else I don’t care

A brief overview of how attacks work and how you detect them would be nice

I know how they work, but how do you detect them?
Consider the simplest of attacks

- stack overflow
function calls revisited

read_buf (buf)
{
    file = open ("my_file");
    read (file, buf, 64)
    close (file)
    return;
}

my_bug ()
{
    int buf [10];
    read_buf (buf);
    print (buf);
    return;
}
all we need to do...

- is load our program in Buf
- and overwrite the return address at 1004 with the begin address of Buf (995)
- and away we go....
how do we detect an attack?
How do we detect an attack?

- detect when EIP loaded with tainted data
- detect when EIP points to tainted data
- detect when certain syscalls have tainted arguments
Skip all of this and take me to signature generation

Tell me more about tracking and forensics

Never mind the tracking, but tell me about forensics
network data tracking
Network Data Tracking

Register = network_read


Memory(A) = Reg. A

Reg.B = Reg.A / 156.345
Network Data Tracking

- Tag network data as "tainted"
Network Data Tracking

- Tag network data as “tainted”
- Track “tainted” data propagation
  - Arithmetic, logical operations
  - Memory operations
Network Data Tracking

- Tag network data as “tainted”
- Track “tainted” data propagation
  - Arithmetic, logical operations
  - Memory operations
- Sanitise data
  - Floating point, SSE
Identifying Attacks

- Jumps
- Function calls
- Function returns
- System calls

EAX EBX ECX EDX RAM

JMP EAX
CALL EAX
RET
JMP A
INT 0x80
Capturing Attacks

- Diverting control flow
- Executing arbitrary instructions
- Overwriting system call arguments
What is the nature of the tags?

- could be simple bit:
  - 1 = tainted,
  - 0 = clean

- or could be offset in network trace
  - single origin
  - full origin
How do we store the tags?

- bitmap (or byte arrays)
  
  101001110101110101001011001001

- pointer arrays
How do we store the tags?

- page tables
How do we store the tags?

- multi-level page tables
Forensics

- When we discover the attack, we do not know the victim
Forensics

- Virtual Address Space
- Applications
  - Process name
  - Linked Libraries
  - Open Ports
- Guest OS
- RAM
- Registers
Signature Generation
Multiple Signature Generators
Logged Network Flows

Critical Exploit Bytes
(e.g. value loaded on EIP)

Argos Memory Log

New Signature

Similar Signatures

Generalised Signature
Signature Generation I

- focus on exploits
  - less mutable than payload
  - used by different payloads
- generates Snort-like signatures
- works well with many existing threats

- true polymorphism is harder…
Signature Generation II

target

URL
Signature Generation II

multiple fields may have contributed to the overflow
Signature Generator II

- handles polymorphic attacks
- easy to check
- great when considering false positives

hard to generate
Signature Generators III and IV

- targeted at encrypted traffic
  - e.g., SSL encrypted connections
  - based on similar principles as previous generators
spare me the details and show some performance numbers

How do you do it, really?
Vulnerability signatures

- static length of the buffer: determine protocol fields which length cannot exceed a given limit
approach: which bytes contributed to the attack?

surprisingly hard!
Let us focus on stack smashing

- heap overflows are also handled but harder to explain
Recall

- stack grows from top to bottom
Recall

- stack grows from top to bottom
How to determine the offending bytes?

easy case

memory

- target
- vulnerable buffer

- tainted data
- untainted data

a protocol field
How to determine the offending bytes?

```c
void a_fun(char *net_buf){
    char vuln_buf[8];
    char a_buffer[8];

    copy net_buf ➔ vuln_buf;
    copy 8B of net_buf ➔ a_buf;
}
```
How to determine the offending bytes?

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    copy net_buf ➔ vuln_buf
    copy 8 bytes of net_buf ➔ unrelated
}
```
we need more info!
First step: AgeStamps

- **AgeStamp**: a global time counter, increased on each function call or return
- whenever tainted value is stored in memory, we associate current AgeStamp with its address
- **stack**: we store history of function calls for each process (stack address + AgeStamp)
- **heap**: we associate current AgeStamp with each heap buffer (malloc interposition)
What can we solve with AgeStamps?

- stale bytes (left by a previous function)

```c
void f(char *net_buf){
    char buf[10];
    strcpy(buf, net_buf);
}
```
What can we solve with AgeStamps?

- stale bytes (left by a previous function)

```c
void f(char *net_buf){
    char buf[10];
    strcpy(buf, net_buf);
}
```
What can we solve with AgeStamps?

- stale bytes (left by a previous function)

Here starts a new function frame with AgeStamp 19

```c
void f(char *net_buf){
    char buf[10];
    strcpy(buf, net_buf);
}
```
What can we solve with AgeStamps?

- stale bytes (left by a previous function)

```c
void a_fun(char *net_buf) {
    char vuln_buf[8];
    char a_buffer[8];

    strcpy(vuln_buf, net_buf + 8);
    strncpy(a_buffer, net_buf, 8);
}
```
But we cannot handle this:

delted gaps (again)

“now: fresh” tainted bytes

⇒ left by *same* function

```c
void a_fun(char *net_buf){
  char unrelated[8];
  char vuln_buf[8];

  while(net_buf[i])
    vuln_buf[++i] = net_buf[++i];

  for(i = 0; i < 8; i++)
    unrelated[i] = net_buf[i];
}
```
we need even more info!
Extra indicators

- Two one-bit indicators:
  - PREV_TAINTED_FRESH
  - PREV_TAINTED_SAVED

- Stored together with AgeStamp, in a memory map,

- Used to track operations that copy tainted data within one function
Example

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[+i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
**Example**

**PREV_TAINTED_FRESH:**
- when `addr` becomes tainted
  \[PTF[addr + 1] := 1,\]
- `(addr + 1): “the memory location below has freshly tainted contents; more recent than my own”`

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++]i;
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

**PREV_TAINTED_FRESH:**
- when `addr` becomes tainted  
  $$\text{PTF}[addr + 1] := 1,$$
- `(addr + 1): “the memory location below has freshly tainted contents; more recent than my own”`

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
**Example**

**PREV_TAINTED_FRESH:**
- when `addr` becomes tainted → `PTF[addr + 1] := 1`,
- `(addr + 1): “the memory location below has freshly tainted contents; more recent than my own”`

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_FRESH:
- when `addr` becomes tainted, \( \text{PTF}[addr + 1] := 1 \),
- \((addr + 1)\): “the memory location below has freshly tainted contents; more recent than my own”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```

<table>
<thead>
<tr>
<th>PREV_TAINTED_FRESH</th>
<th>PREV_TAINTED_SAVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20 1</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

- **tainted data**
- **untainted data**
- **unrelated tainted data**

Herbert Bos  
VU Amsterdam

---

*Example*
Example

PREV_TAINTED_SAVED:
when \( addr \) becomes tainted, 
and \( \text{PTF}[addr] = 1 \),
\( \Rightarrow \text{PTS}[addr] := 1 \), and \( \text{PTF}[addr] := 0 \)
(addr): "my current value is the result of the first tainted store op since \( (addr-1) \) became tainted"

```
void a_fun(char *net_buf){
  char unrelated[8];
  char vuln_buf[8];
  while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
  for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when *addr* becomes tainted,
and PTF[*addr*] = 1,
⇒ PTS[*addr*] := 1, and PTF[*addr*] := 0
(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++] = net_buf[++];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED: when \textit{addr} becomes tainted, and PTF[\textit{addr}] = 1,
\(\Rightarrow\) PTS[\textit{addr}] := 1, and PTF[\textit{addr}] := 0
(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

```c
void a_fun(char *net_buf){
  char unrelated[8];
  char vuln_buf[8];
  while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
  for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
**Example**

PREV_TAINTED_SAVED:
when \( addr \) becomes tainted,
and \( \text{PTF}[addr] = 1 \),
\( \Rightarrow \text{PTS}[addr] := 1 \), and \( \text{PTF}[addr] := 0 \)
(addr): “my current value is the result of the first tainted store op since \( (addr-1) \) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```

PREV_TAINTED_SAVED
PREV_TAINTED_FRESH
unrelated
vuln_buf

- Green: tainted data
- Yellow: untainted data
- Blue: unrelated tainted data
PREV_TAINTED_SAVED:
when \( addr \) becomes tainted, and \( \text{PTF}[addr] = 1 \),
\( \rightarrow \text{PTS}[addr] := 1 \), and \( \text{PTF}[addr] := 0 \)
(addr): “my current value is the result of the first tainted store op since \((addr-1)\) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when \textit{addr} becomes tainted, 
\textbf{and} \( \text{PTF}[\text{addr}] = 1 \), 
\( \Rightarrow \text{PTS}[\text{addr}] := 1 \), \textbf{and} \( \text{PTF}[\text{addr}] := 0 \) 
\( \text{(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”} \)

void a\_fun(char *net\_buf){
    char unrelated[8];
    char vuln\_buf[8];
    while(net\_buf[i]) vuln\_buf[++i] = net\_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net\_buf[i];
}
Example

PREV_TAINTED_SAVED:
when \textit{addr} becomes tainted, 
\textbf{and} PTF[\textit{addr}] = 1, 
\rightarrow PTS[\textit{addr}] := 1, \textbf{and} PTF[\textit{addr}] := 0 
(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

\begin{verbatim}
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
\end{verbatim}
Example

PREV_TAINTED_SAVED:
when $addr$ becomes tainted, and $\text{PTF}[addr] = 1$,
$\Rightarrow \text{PTS}[addr] := 1$, and $\text{PTF}[addr] := 0$
(addr): “my current value is the result of the first tainted store op since $(addr-1)$ became tainted”

```
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED: when *addr* becomes tainted, and PTF[*addr*] = 1, 
\[\Rightarrow\] PTS[*addr*] := 1, and PTF[*addr*] := 0

(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

```
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when $addr$ becomes tainted, and $\text{PTF}[addr] = 1$,
$\Rightarrow \text{PTS}[addr] := 1$, and $\text{PTF}[addr] := 0$

(addr): “my current value is the result of the first tainted store op since $(addr-1)$ became tainted”

```
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when addr becomes tainted,
and PTF[addr] = 1,
⇒ PTS[addr] := 1, and PTF[addr] := 0
(addr): "my current value is the result
of the first tainted store op since
(addr-1) became tainted"

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when $addr$ becomes tainted,
and $\text{PTF}[addr] = 1$,
$\rightarrow \text{PTS}[addr] := 1$, and $\text{PTF}[addr] := 0$

(addr): “my current value is the result of the first tainted store op since $(addr-1)$ became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
**Example**

PREV_TAINTED_SAVED:
when \( addr \) becomes tainted,
and \( \text{PTF}[addr] = 1 \),
\( \Rightarrow \text{PTS}[addr] := 1 \), and \( \text{PTF}[addr] := 0 \)
(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED: when \( addr \) becomes tainted, and \( \text{PTF}[addr] = 1 \),
\( \Rightarrow \) \( \text{PTS}[addr] := 1 \), and \( \text{PTF}[addr] := 0 \)
(addr): “my current value is the result of the first tainted store op since \((addr-1)\) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when \texttt{addr} becomes tainted, \textbf{and} \texttt{PTF[addr]} = 1,
\[\Rightarrow\texttt{PTS[addr]} := 1, \textbf{and} \texttt{PTF[addr]} := 0\]
\(\texttt{(addr)}): \text{“my current value is the result of the first tainted store op since (addr-1) became tainted”}\)

void a\_fun(char *\texttt{net\_buf}){
    char \texttt{unrelated}[8];
    char \texttt{vuln\_buf}[8];

    while(\texttt{net\_buf}[i]) \texttt{vuln\_buf}[++i] = \texttt{net\_buf}[++i];
    for(i = 0; i < 8; i++) \texttt{unrelated}[i] = \texttt{net\_buf}[i];
}
Example

PREV_TAINTED_SAVED:
when $addr$ becomes tainted, and $PTF[addr] = 1$,
$PTS[addr] := 1$, and $PTF[addr] := 0$
(addr): “my current value is the result of the first tainted store op since $(addr-1)$ became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
**Example**

**PREV_TAINTED_SAVED:**
when `addr` becomes tainted, 
and `PTF[addr] = 1`,
⇒ `PTS[addr] := 1`, and `PTF[addr] := 0` (addr): “my current value is the result of the first tainted store op since `(addr-1)` became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when \( addr \) becomes tainted, 
\[ \text{and PTF}[addr] = 1, \]
\[ \Rightarrow \text{PTS}[addr] := 1, \text{and PTF}[addr] := 0 \]
(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when \textit{addr} becomes tainted,
\textbf{and} \text{PTF}[\textit{addr}] = 1,
\textbf{⇒} \text{PTS}[\textit{addr}] := 1, \textbf{and} \text{PTF}[\textit{addr}] := 0

(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

```
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
### Example

**PREV_TAINTED_SAVED:**
when \( addr \) becomes tainted, and \( \text{PTF}[addr] = 1 \),

\[ \Rightarrow \text{PTS}[addr] := 1, \text{and} \ \text{PTF}[addr] := 0 \]

\( (addr) \): “my current value is the result of the first tainted store op since \( (addr-1) \) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when \textit{addr} becomes tainted,
\textbf{and} \text{PTF}[addr] = 1,
\Rightarrow \text{PTS}[addr] := 1, \textbf{and} \text{PTF}[addr] := 0

(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

\begin{verbatim}
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
\end{verbatim}
Example

PREV_TAINTED_SAVED:
when `addr` becomes tainted,
and \( \text{PTF}[addr] = 1 \),
\( \text{PTS}[addr] := 1 \), and \( \text{PTF}[addr] := 0 \)
(addr): “my current value is the result of the first tainted store op since \((addr-1)\) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++] = net_buf[++];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```

<table>
<thead>
<tr>
<th></th>
<th>PREV_TAINTED_SAVED</th>
<th>PREV_TAINTED_FRESH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20 0 1</td>
<td>20 0 1</td>
</tr>
<tr>
<td></td>
<td>20 1 1</td>
<td>20 1 1</td>
</tr>
<tr>
<td></td>
<td>20 0 1</td>
<td>20 0 0</td>
</tr>
<tr>
<td></td>
<td>20 0 1</td>
<td>20 0 1</td>
</tr>
<tr>
<td></td>
<td>20 0 1</td>
<td>20 0 1</td>
</tr>
<tr>
<td></td>
<td>20 0 1</td>
<td>20</td>
</tr>
</tbody>
</table>

- **tainted data**
- **untainted data**
- **unrelated tainted data**
- **unrelated data**
Example

PREV_TAINTED_SAVED:
when addr becomes tainted,
and PTF[addr] = 1,
⇒ PTS[addr] := 1, and PTF[addr] := 0
(addr): “my current value is the result of the first tainted store op since (addr-1) became tainted”

Example

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when \( addr \) becomes tainted, and \( PTF[addr] = 1 \),
\( \Rightarrow PTS[addr] := 1, \text{ and } PTF[addr] := 0 \)
(addr): “my current value is the result of the first tainted store op since \((addr - 1)\) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when \( addr \) becomes tainted, and \( PTF[addr] = 1 \),
\( PTS[addr] := 1 \), and \( PTF[addr] := 0 \)
(\( addr \)): “my current value is the result
of the first tainted store op since
(\( addr-1 \)) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when $addr$ becomes tainted, and $PTF[addr] = 1$,
$PTS[addr] := 1$, and $PTF[addr] := 0$
(addr): “my current value is the result of the first tainted store op since $(addr-1)$ became tainted”

void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
Example

PREV_TAINTED_SAVED:
when \( addr \) becomes tainted, and \( \text{PTF}[addr] = 1 \),
\( \Rightarrow \text{PTS}[addr] := 1 \), and \( \text{PTF}[addr] := 0 \)
(addr): “my current value is the result of the first tainted store op since \( (addr-1) \) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

**PREV_TAINTED_SAVED:**

when \( addr \) becomes tainted, **and** \( PTF[addr] = 1 \),

\[ PTS[addr] := 1, \text{ and } PTF[addr] := 0 \]

(addr): “my current value is the result of the first tainted store op since \((addr-1)\) became tainted”

```c
void a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];
    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
Example

PREV_TAINTED_SAVED:
when $addr$ becomes tainted,
and $\text{PTF}[addr] = 1$,
$\Rightarrow \text{PTS}[addr] := 1$, and $\text{PTF}[addr] := 0$

$(addr)$: “my current value is the result of the first tainted store op since $(addr-1)$ became tainted”

```c
def a_fun(char *net_buf){
    char unrelated[8];
    char vuln_buf[8];

    while(net_buf[i]) vuln_buf[++i] = net_buf[++i];
    for(i = 0; i < 8; i++) unrelated[i] = net_buf[i];
}
```
In summary…

- to discover tainted gap formed in the same epoch as offending memory region:
  - byte above the top gap byte has both indicators set to 1,
  - bottom gap byte has both indicators set to 0,
  - (Easy) observation: if two tainted gaps are merged, the above characteristics holds.
As a result, signatures are easy
Limitations

- “fresh” tainted bytes, left by the same function, copied in a correct order

```c
void a_fun(char *net_buf){
    char vuln_buf[8];
    char unrelated[8];

    for(i = 0; i < 8; i++)
        unrelated[i] = net_buf[i];

    while(net_buf[i])
        vuln_buf[++i] = net_buf[++i];
}
```
On the other hand…

- even if this extra data forms a protocol field, including it in a signature won't cause false positives, only false negatives

\[ L_1 = \text{size}\text{(unrelated)} \]
\[ L_2 > \text{size}\text{(vuln\_buf)} \]

Any message with these two fields having length greater than \((L_1 + L_2)\) will overflow the buffer in memory.
Argos Overhead
Emulator Performance

Overhead (y times slower)

![Bar chart showing overhead for various applications: bunzip2, apache, nbench integer, nbench float, nbench memory. The chart compares Vanilla Qemu and Argos.]
Signature Generation Performance

Time to generate signature (sec)

Tcpdump trace size (MB)
I cannot take anymore, go away (tell me about applications)

Tell me about good spirits
how to detect new attacks quickly?
Answer: we need Guinea Pigs

- Honeypots
- Designed to get infected
- Analyse attacks as they occur
This should be effortless…
Argos: advertised honeypots

- full system emulator
- detect attacks by dynamic taint analysis

spray more, get more
Honeypots have problems…
honeypot: detect and avoid?
configuration problems

spot the differences
more work: control + mgmt
Limited coverage

- worse: dark IP space
geared to servers
Solution: Eudaemon

- change every machine into a honeypot when needed

*spray more, get more*
Eudaemon: A Good Spirit

whoopie!
Examples

- honeypot “screensaver”
Examples

- “Big Red Button”
Eudaemon

- Merge desktop PCs and honeypots
- Minimize user discomfort
  - Using idle cycles
  - On user demand
  - Transparent

www.suspicious.net

Eudaemon as Big Red Button
Eudaemon as Screensaver
Main idea

- Implement ‘Argos’ as userspace library
- Make every process link with emulator
- When possessing:
  - attach to a process (e.g., using ptrace)
  - capture processor state
  - inject shell code, and create space for emulator
  - inject state in emulator and start
  - away you go!
that is quite enough, thank you

how about performance

Whoa, what would I do without details?

eudaemon details
Eudaemon Technical Overview

Process Address Space

- Binary (aka text)
- Heap
- Shared libraries
- Stack

LD_PRELOAD=libemulator.so

enter_emulation(process state)
exit_emulation()
Emulator Library

- Based on qemu-user
- Intrusion detection using dynamic taint analysis (Argos)
- Tainted data origins
  - Network (reading from socket descriptors)
  - IPC (shared memory, message queues)
  - Unsafe files (user and temporary files)
Eudaemon Possession: Phase 1

- **Binary (aka text)**
- **Shared libraries**
- **ELF header**
- **Heap**
- **Process state (registers, MMX)**

**Eudaemon**
- **Activation Shellcode**

**Target Process**
- **ptrace(ATTACH)**
- **Inject shellcode**

**Discover library symbols**

**Argos library**

**Shellcode**

**Process state**

Herbert Bos  
VU Amsterdam
Eudaemon Possession: Phase 2

- Activation
  - Shellcode
    - emulator_run(state)
    - trap
    - mmap()
- Target Process
  - ELF header
  - Binary (aka text)
  - Heap
  - Emulator stack
  - Shared libraries
  - Argos library
  - Stack
  - Process state (registers, MMX)

- Detach

Argos library
Eudaemon Release:
Phase 1

- ELF header
- Binary (aka text)
- Heap
- Emulator stack
- Shared libraries
- Argos library
- Stack
- Process state (registers, MMX)

**Flow Diagram:**

1. Eudaemon
2. Argos library
3. Deactivation
4. Shellcode
5. ptrace(ATTACH)
6. Inject shellcode
7. Discover library symbols
8. Save process state
9. Target Process
Eudaemon Release:
Phase 1

Eudaemon

- Deactivation
- Shellcode

Argos library

Target Process

- ELF header
- Binary (aka text)
- Heap
- Emulator stack

- Shared libraries
- Argos library
- Stack
- Process state (registers, MMX)

emulator_stop() trap

Restore state

trap

 Herbert Bos

VU Amsterdam

140
Eudaemon Release: Phase 2

Target Process
- ELF header
- Binary (aka text)
- Heap
- Emulator stack
- Shared libraries
- Argos library
- Stack
- Process state (registers, MMX)

Eudaemon
- Deactivation
- Shellcode
- Argos library

Detach
- mmap()
- emulator_run()
- trap
- Retrieve emulator state
- munmap()
- emulator_stop()
- trap
- Restore Process state

Herbert Bos VU Amsterdam
Eudaemon performance
Eudaemon
Performance Evaluation

High-load Possession

Low-load Possession
## Emulator Performance Evaluation

*File used is linux-2.6.14.5.tar.bz2*

<table>
<thead>
<tr>
<th>Application</th>
<th>Native Execution</th>
<th>Emulated Execution</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>bunzip2</td>
<td>27.99 sec</td>
<td>230.44 sec</td>
<td>723.30%</td>
</tr>
<tr>
<td>wget</td>
<td>10.73 MB/s</td>
<td>10.75 MB/s</td>
<td>0%</td>
</tr>
</tbody>
</table>
Eudaemon conclusions

- Promising approach for collaborative attack detection
- Acceptable performance
- Novel
- Implementation not rich enough yet 😞
Argos downloaded 1000+ times

- SURFnet
- Eurecom
- Nepenthes
- Forth
- DFN CERT
- ETH Zurich
- TNO
to conclusions

example applications
Applications in more detail
LONT

- we connected Argos to TNO’s LONT
  - Argos detects attack
  - Generates signature
LONT

- connected to TNO’s LONT application
  - Argos detects attack
  - Generates signature
  - LONT monitors the spread of the attack
SURFnet

- detect new and known attacks
- many sensors

Herbert Bos
SURFnet

- detect new and known attacks
- many sensors
GovCERT

A WORD FROM THE EXECUTIVE DIRECTOR

To mark the European Union’s (EU) 50th birthday we have recently witnessed a

Prof. Posch commented: “I would like to extend my gratitu...
Starting up an Early Warning System in the Netherlands

Menno Muller

The aim of the Dutch government is to have 65% of all its services to citizens available on-line by the end of 2007. This is expected to increase significantly the number of electronic transactions between citizens and governmental systems. Monitoring the relevant data-flows is a daunting task. However, from a security perspective, it is vital to keep an eye on what is going on. This increased data traffic demands smarter monitoring tools for the organisations involved.

GOVCERT.NL is the Computer Emergency Response Team (CERT) for the Dutch

As Nepenthes depends heavily on software components that simulate vulnerabilities, GOVCERT.NL aims to develop new functionality. The focus is on finding new ways to detect unknown exploits. One initiative in this area is the integration of Argos (www.few.vu.nl/argos/) into the monitoring system. Argos is an emulator for capturing zero-day attacks and is currently being developed by the Vrije University (VU) at Amsterdam. Exploring the possibilities of integration of Argos is a joint project with SURFnet. Another way of integrating Argos could be to implement a pre-processor that incorporates a shell-code detection technique. As the best solution has not yet been found, several alternative approaches are being investigated.

Lessons learned

For example, the Network Monitoring Special Interest Group for the Forum for connected to the Internet that does not serve any purpose other than attracting
The EU FP6 NoAH architecture
ScriptGen

![Diagram of SGNET architecture with nodes labeled SG1, SG2, SG3, SF1, SF2, SF3, SH1, SH2, and GW connected within a private network.]

Figure 2: SGNET architecture
Forth

- Shadow Honeypots

Internet

white list
Conclusions

- What did I say?!
- Dynamic taint analysis quite powerful detection technique
  - most common attack vectors
  - worms
- Mostly suitable for honeypots
  - too slow for full-time protection of all processes
  - but perhaps apply when needed
Future Work

- Replaying attacks
- Integration with nepenthes honeypot
  - ✔ Increase data tracking precision
  - ✔ Protocol aware signature generation
- Generate self certifying alerts
On The Web

http://www.few.vu.nl/argos

Argos: an Emulator for Capturing Zero-Day Attacks

Argos is a full and secure system emulator designed for use in Honeypots. It is based on QEMU, an open source processor emulator that uses dynamic translation to achieve a fairly good emulation speed.

We have extended QEMU to enable it to detect remote attempts to compromise the emulated guest operating system. Using dynamic taint analysis Argos tracks network data throughout the processor's execution and detects any attempts to use them in a malicious way. When an attack is detected the memory footprint of the attack is logged and the emulators exits.

Argos is the first step to create a framework that will use next generation honeypots to automatically identify and produce remedies for zero-day worms, and other similar attacks. Next generation honeypots should not require that the honeypot's IP address remain un-advertised. On the contrary, it should attempt to publicize its services and even actively generate traffic. In former honeypots this was often impossible, because malicious and benevolent traffic could not be distinguished. Since Argos is explicitly signaling each possibly successful exploit attempt, we are now able to differentiate malicious attacks and innocuous traffic.

Argos has moved to the local gForge. You can pickup the code and documentation there.
draw some conclusions
SweetBait Design
## Logs Format

<table>
<thead>
<tr>
<th>Format</th>
<th>Type</th>
<th>RID</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register values</td>
<td></td>
<td></td>
<td>Register tags</td>
</tr>
<tr>
<td>EIP value</td>
<td>EIP origin</td>
<td>EFLAGS</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format</th>
<th>Tainted Flag</th>
<th>Size</th>
<th>P. Address</th>
<th>V. Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Block Contents</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Forensics Shellcode Injection

- Lookup process’s read-only pages
- Inject code at last text segment page
- Point EIP to shellcode
Forensics – The Snitch

- Pid = getpid()
- Rid [injected by Argos]
- Connect(localhost)
- Send(pid & rid)
- Listen()
- Accept()
- Read(pid & rid)
- Exec(Netstat or OpenPorts)
- Connect(argos host)
- Send(info)
**Legend**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_t$</td>
<td>address of target</td>
</tr>
<tr>
<td>target</td>
<td>target address/return</td>
</tr>
<tr>
<td>$A$</td>
<td>address to store in target</td>
</tr>
<tr>
<td>$N_t$</td>
<td>offset in network trace of the bytes that overwrite the target</td>
</tr>
</tbody>
</table>

This address will overwrite target.

Repeat address to handle different offsets.

Network trace:

$\text{L}_1$ (full protocol field)