The worm threat

- Worms are a serious threat
- Worm propagation disrupts Internet traffic
- Attacker gains control of infected machines
- Worms spread too fast for human response
- Slammer scanned most of the Internet in 10 minutes, infected 90% of vulnerable hosts
- Worm containment must be automatic
Automatic worm containment

- Previous solutions are network centric
  - analyze network traffic
  - generate signature and drop matching traffic or
  - block hosts with abnormal network behavior

- No vulnerability information at network level
  - false negatives: worm traffic appears normal
  - false positives: good traffic misclassified

**false positives are a barrier to automation**
Vigilante's end-to-end architecture

- Host-based detection
  - instrument software to analyze infection attempts
- Cooperative detection without trust
  - detectors generate self-certifying alerts (SCAs)
  - detectors broadcast SCAs
- Hosts generate filters to block infection

*Contains fast spreading worms: no false positives, deployable today*
The Outline

- Detection
- Self-certifying alerts (SCAs)
- Generation of vulnerability-specific filters
- Evaluation
Detection

"Worms? She must have picked them up on the Internet."
Detection

- Non-executable pages
- Dynamic dataflow analysis
Detection: Non-executable pages

- Uses non-execute protection on stack and heap pages to detect and prevent code injection attacks.
- It has negligible runtime overhead with emerging hardware support and has relatively low overhead even when emulated in software.
- Can be used to generate arbitrary execution control or arbitrary code execution SCAs as follows:
  - **When the worm attempts to execute code in a protected page, an exception is thrown.**
Detection: Dynamic dataflow analysis

- Track the flow of data from input messages
- Mark memory as dirty when data is received
- Track all data movement
- Trap the worm before it executes any instructions
- Trap execution of dirty data
- Trap loading of dirty data into program counter
If dirty data is about to be loaded into the program counter, it signals an attempt to exploit an arbitrary execution control vulnerability.

If dirty data is about to be executed, it signals an attempt to exploit an arbitrary code execution vulnerability.

If a critical argument to a critical function is dirty, it signals an attempt to exploit an arbitrary function argument vulnerability.
Whenever an instruction that moves data from a source to a destination is executed, the destination becomes **dirty if the source is dirty** and becomes clean otherwise.

When a destination becomes dirty, it is tagged with the identifier associated with the source.

Whenever data is received from a network connection, the memory locations where the data is written are **marked dirty and tagged** with sequence numbers corresponding to each received byte.
Infection Attempts!!!

- The *instrumented control flow instructions* signal an infection attempt when *dirty data is about to be executed or loaded* into the program counter.

- The *instrumented critical functions* signal an infection attempt when all the *bytes in a critical argument* are dirty.
Dynamic dataflow analysis

//vulnerable code
push len
push netbuf
push sock
call recv
push netbuf
push localbuf
call strcpy
ret

Since the copy overwrites the return address in the stack, return attempts to load dirty data into the program counter. Detector generates an arbitrary execution control alert

- alert: value loaded into program counter is dirty
- high-coverage: stack, function pointers, ...
Dynamic dataflow analysis

- works with normal binaries
- instrumentation at runtime

normal .exe

detection

vulnerable process
Where are the detectors?

- General detectors are expensive
- Centralized detectors can be attacked
- Any host can be a detector
  - load sharing, high coverage, resilience
- Detectors create self-certifying alerts
Self-Certifying Alerts
Self-certifying alerts

- Machine-verifiable proofs of vulnerability
  - identify an application and a type of vulnerability
  - contain log of attack messages
  - contain verification information
- Enable hosts to verify if they are vulnerable
  - replay infection with modified messages
  - verification has no false positives
SCA types

- Arbitrary code execution (ACE)
- Arbitrary execution control (AEC)
- Arbitrary function argument (AFA)

**What can the attacker do?** inject code

**What is the verification information?** code location
SCA types

- Arbitrary code execution (ACE)
- Arbitrary execution control (AEC)
- Arbitrary function argument (AFA)

What can the attacker do? force a control flow transfer

What is the verification information? location of program counter
SCA types

- Arbitrary code execution (ACE)
- Arbitrary execution control (AEC)
- Arbitrary function argument (AFA)

What can the attacker do? supply an argument to a function

What is the verification information? function name & location of argument.

alert type: AFA
attack messages:
verification information: ...

Virginia Tech
SCA generation

- Log messages
- Generate SCA when worm is detected
  - search log for relevant messages
  - compute verification information
  - generate tentative version of SCA
  - repeat until verification succeeds
- Detectors may guide search
Generating an AEC alert

//vulnerable code
push len
push netbuf
push sock
call recv
push netbuf
push localbuf
call strcpy
ret

log: msg1 1111111111111111111

SCA: AEC, 1111111111111111111, pc at offset 136
Verifying an AEC alert

alert type: Arbitrary Execution Control
attack message: 1111111111111111
verification information: pc at offset 6 of message

proves that external interfaces allow arbitrary control of the execution

verification is independent of detection mechanism
SCA broadcast

- Uses overlay of superpeers
  - Akamai-like overlay with added security
  - Detectors flood alerts over overlay links
- Denial-of-service prevention
  - Per-link rate limiting
  - Per-hop filtering and verification
- Controlled disclosure of overlay membership

Hosts receive SCAs with high probability
Protection (Filters)
Protection

- Hosts generate filter from SCA
- Mutations make protection difficult (as in real diseases)

**attack:**
```
add eax,1; mov ebx, eax
```

| 0x3 | 0x24 | 0x67 | 0x42 | 0x1 |

**mutation:**
```
inc eax; push eax; pop ebx
```

| 0x3 | 0x12 | 0x28 | 0x63 | 0x4 |
Filter generation

- **Dynamic data and control flow analysis**
  - track control and data flow from input messages
  - compute conditions that determine execution path
  - filter blocks messages that satisfy conditions
- Uses full data flow information
  - dataflow graphs for dirty data and CPU flags
  - record decisions on conditional instructions
Generating filters for vulnerabilities

//vulnerable code
recv msg
mov al, [msg]
mov cl, 0x3
cmp al, cl
jne L2  //msg[0] == 3 ?
xor eax, eax
L1 mov [esp+eax+4], cl
mov cl, [eax+msg+1]
inc eax
test cl, cl
jne L1  //msg[i] == 0 ?
L2 ret

attack: 0x3 0x24 0x67 0x42 0x1
filter: =3 ≠0 ≠0 ≠0 ≠0
mutation: 0x3 0x12 0x28 0x63 0x4

Match!
Filters as program slices

 Filters are a subset of the program’s instructions

```assembly
//recv msg
mov al,[msg]
mov cl,0x3
cmp al,cl
jne L2  //msg[0] == 3 ?
xor eax,eax
mov [esp+eax+4],cl
mov cl,[eax+msg+1]
inc eax
test cl,cl
jne L1  //msg[i] == 0 ?
ret

//test msg[1],msg1[1]; je out
```

```assembly
//recv msg
mov al,[msg]
mov cl,0x3
cmp al,cl
jne L2  //msg[0] == 3 ?
xor eax,eax
mov [esp+eax+4],cl
mov cl,[eax+msg+1]
inc eax
test cl,cl
jne L1  //msg[i] == 0 ?
ret
```
Filters

- Capture generic conditions
- Safe and efficient: no side effects, no loops
- Two-filter design reduces false negatives
  - a **specific filter** without false positives, and a **general filter** that may have false positives but matches more messages than the specific filter to block more worm variants.
Summing it Up

Network

Detection Engine
SCA Generation
SCA Verification
SCA Distribution
Detector Host

Network

Filter
Vulnerable Application
Protection
SCA Verification
SCA Distribution
Vulnerable Host

Network
Evaluation
Evaluation

- Three real worms:
  - Slammer (SQL server), Blaster (RPC), CodeRed (IIS)
- Measurements of prototype implementation
- SCA generation and verification
- Filter generation
- Filtering overhead
- Simulations of SCA propagation with attacks
Slammer infected approximately 75,000 Microsoft SQL Servers. It was the fastest computer worm in history.

During its outbreak, the number of infected machines doubled every 8.5 seconds.

Slammer’s exploit uses a UDP packet with the first byte set to 0x04 followed by a 375 byte string with the worm code. While copying the string, SQL overwrites a return address in the stack.
CodeRed infected approximately 360,000 Microsoft IIS servers. It spread much slower than Slammer, taking approximately 37 minutes to double the infected population.

CodeRed’s exploit sends a “GET /default.ida?” request followed by 224 ‘X’ characters, the URL encoding of 22 Unicode characters (with the form “%uHHHH” where H is an hexadecimal digit), “HTTP/1.0”, headers and an entity body with the worm code.

While processing the request, IIS overwrites the address of an exception handler with a value derived from the ASCII encoding of the Unicode characters. The worm gains control by triggering an exception in a C runtime function and it immediately transfers control to the main worm code that is stored in the heap.
Blaster infected the RPC service on Microsoft Windows machines. We conservatively estimate that it infected 500,000 hosts and that its spread rate was similar to CodeRed’s.

Blaster is a **two-message** attack: the first message is an DCERPC **bind request** and the second is a DCERPC DCOM **object activation request**. The second message has a field that contains a network path starting with ‘\\’. While copying this field to a buffer and searching for a terminating ‘\’, the **RPC service overwrites a return address in the stack**.
Time to generate SCAs

<table>
<thead>
<tr>
<th></th>
<th>SCA generation time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slammer</td>
<td>18</td>
</tr>
<tr>
<td>Blaster</td>
<td>206</td>
</tr>
<tr>
<td>CodeRed</td>
<td>2667</td>
</tr>
</tbody>
</table>
Time to verify SCAs

![Bar chart showing SCA verification times for different malicious code types: Slammer (10 ms), Blaster (18 ms), CodeRed (75 ms)]](image)
Time to generate filters

<table>
<thead>
<tr>
<th>Filter</th>
<th>Filter generation time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slammer</td>
<td>24</td>
</tr>
<tr>
<td>Blaster</td>
<td>273</td>
</tr>
<tr>
<td>CodeRed</td>
<td>3402</td>
</tr>
</tbody>
</table>
Simulating SCA propagation

- Susceptible/Infective epidemic model
- 500,000 node network on GeorgiaTech topology
- Network congestion effects
  - RIPE data gathered during Slammer’s outbreak
  - delay/loss increase linearly with infected hosts
- DoS attacks
  - infected hosts generate fake SCAs
  - verification increases linearly with number of SCAs
Conclusion

- Vigilante can contain worms automatically
- Requires no prior knowledge of vulnerabilities
- No false positives
- Low false negatives
- Deployable today
Thanks for your attention

Questions Welcomed

Regards,
Hussein M. Ahmed