CS 155 Spring 2006

Network Worms: Attacks and Defenses

John Mitchell

with slides borrowed from various (noted) sources

Outline

- Worm propagation
  - Worm examples
  - Propagation models
- Detection methods
  - Traffic patterns: EarlyBird
  - Watch attack: TaintCheck and Sting
  - Look at vulnerabilities: Generic Exploit Blocking
- Disable
  - Generate worm signatures and use in network or host-based filters

2

Worm

- A worm is self-replicating software designed to spread through the network
  - Typically exploit security flaws in widely used services
  - Can cause enormous damage
    - Launch DDOS attacks, install bot networks
    - Access sensitive information
    - Cause confusion by corrupting the sensitive information
- Worm vs Virus vs Trojan horse
  - A virus is code embedded in a file or program
  - Viruses and Trojan horses rely on human intervention
  - Worms are self-contained and may spread autonomously

3

Cost of worm attacks

- Morris worm, 1988
  - Infected approximately 6,000 machines
    - 10% of computers connected to the Internet
  - cost ~ $10 million in downtime and cleanup
- Code Red worm, July 16 2001
  - Direct descendant of Morris' worm
  - Infected more than 500,000 servers
    - Programmed to go into infinite sleep mode July 28
  - Caused ~ $2.6 Billion in damages,
- Love Bug worm: $8.75 billion

4

Statistics: Computer Economics Inc., Carlsbad, California

Aggregate statistics

Financial Impact of Virus Attacks 1995—2005

	Worldwide Impact (US $)
2005	$14.2 Billion
2004	17.5 Billion
2003	13.0 Billion
2002	11.1 Billion
2001	13.2 Billion
2000	17.1 Billion
1999	13.0 Billion
1998	6.1 Billion
1997	3.3 Billion
1996	1.8 Billion
1995	500 Million

Source: Computer Economics, 2006 *Figure 1*

5

Internet Worm (First major attack)

- Released November 1988
  - Program spread through Digital, Sun workstations
  - Exploited Unix security vulnerabilities
    - VAX computers and SUN-3 workstations running versions 4.2 and 4.3 Berkeley UNIX code
- Consequences
  - No immediate damage from program itself
  - Replication and threat of damage
    - Load on network, systems used in attack
    - Many systems shut down to prevent further attack

6

Internet Worm Description

- Two parts
  - Program to spread worm
    - look for other machines that could be infected
    - try to find ways of infiltrating these machines
  - Vector program (99 lines of C)
    - compiled and run on the infected machines
    - transferred main program to continue attack
- Security vulnerabilities
  - fingerd – Unix finger daemon
  - sendmail - mail distribution program
  - Trusted logins (.rhosts)
  - Weak passwords

7

Three ways the worm spread

- Sendmail
  - Exploit debug option in sendmail to allow shell access
- Fingerd
  - Exploit a buffer overflow in the fgets function
  - Apparently, this was the most successful attack
- Rsh
  - Exploit trusted hosts
  - Password cracking

8

sendmail

- Worm used debug feature
  - Opens TCP connection to machine's SMTP port
  - Invokes debug mode
  - Sends a RCPT TO that pipes data through shell
  - Shell script retrieves worm main program
    - places 40-line C program in temporary file called x$$,l1.c where $$ is current process ID
    - Compiles and executes this program
    - Opens socket to machine that sent script
    - Retrieves worm main program, compiles it and runs

9

fingerd

- Written in C and runs continuously
- Array bounds attack
  - Fingerd expects an input string
  - Worm writes long string to internal 512-byte buffer
- Attack string
  - Includes machine instructions
  - Overwrites return address
  - Invokes a remote shell
  - Executes privileged commands

10

Remote shell

- Unix trust information
  - /etc/host.equiv – system wide trusted hosts file
  - /.rhosts and ~/.rhosts – users' trusted hosts file
- Worm exploited trust information
  - Examining files that listed trusted machines
  - Assume reciprocal trust
    - If X trusts Y, then maybe Y trusts X
- Password cracking
    - Worm was running as daemon (not root) so needed to break into accounts to use .rhosts feature
    - Dictionary attack
    - Read /etc/passwd, used ~400 common password strings

11

The worm itself

- Program is called 'sh'
  - Clobbers argv array so a 'ps' will not show its name
  - Opens its files, then unlinks (deletes) them so can't be found
    - Since files are open, worm can still access their contents
- Tries to infect as many other hosts as possible
  - When worm successfully connects, forks a child to continue the infection while the parent keeps trying new hosts
- Worm did not:
  - Delete system's files, modify existing files, install trojan horses, record or transmit decrypted passwords, capture superuser privileges, propagate over UUCP, X.25, DECNET, or BITNET

12

Detecting Morris Internet Worm

- Files
  - Strange files appeared in infected systems
  - Strange log messages for certain programs
- System load
  - Infection generates a number of processes
  - Systems were reinfected => number of processes grew and systems became overloaded
    - Apparently not intended by worm's creator

Thousands of systems were shut down

13

Stopping the worm

- System admins busy for several days
  - Devised, distributed, installed modifications
- Perpetrator
  - Student at Cornell; discovered quickly and charged
  - Sentence: community service and $10,000 fine
    - Program did not cause deliberate damage
    - Tried (failed) to control # of processes on host machines
- Lessons?
  - Security vulnerabilities come from system flaws
  - Diversity is useful for resisting attack
  - "Experiments" can be dangerous

14

Sources for more information

- Eugene H. Spafford, The Internet Worm: Crisis and Aftermath, CACM 32(6) 678-687, June 1989
- Page, Bob, "A Report on the Internet Worm", http://www.ee.ryerson.ca:8080/~elf/hack/iworm.html

15

Some historical worms of note

Worm	Date	Distinction
Morris	11/88	Used multiple vulnerabilities, propagate to "nearby" sys
ADM	5/98	Random scanning of IP address space
Ramen	1/01	Exploited three vulnerabilities
Lion	3/01	Stealthy, rootkit worm
Cheese	6/01	Vigilante worm that secured vulnerable systems
Code Red	7/01	First sig Windows worm; Completely memory resident
Walk	8/01	Recompiled source code locally
Nimda	9/01	Windows worm: client-to-server, c-to-c, s-to-s, …
Scalper	6/02	11 days after announcement of vulnerability; peer-to-peer network of compromised systems
Slammer	1/03	Used a single UDP packet for explosive growth

Kienzle and Elder

16

Increasing propagation speed

- Code Red, July 2001
  - Affects Microsoft Index Server 2.0,
    - Windows 2000 Indexing service on Windows NT 4.0.
    - Windows 2000 that run IIS 4.0 and 5.0 Web servers
  - Exploits known buffer overflow in Idq.dll
  - Vulnerable population (360,000 servers) infected in 14 hours
- SQL Slammer, January 2003
  - Affects in Microsoft SQL 2000
  - Exploits known buffer overflow vulnerability
    - Server Resolution service vulnerability reported June 2002
    - Patched released in July 2002 Bulletin MS02-39
  - Vulnerable population infected in less than 10 minutes

17

Code Red

- Initial version released July 13, 2001
  - Sends its code as an HTTP request
  - HTTP request exploits buffer overflow
  - Malicious code is not stored in a file
    - Placed in memory and then run
- When executed,
  - Worm checks for the file C:\Notworm
    - If file exists, the worm thread goes into infinite sleep state
  - Creates new threads
    - If the date is before the 20th of the month, the next 99 threads attempt to exploit more computers by targeting random IP addresses

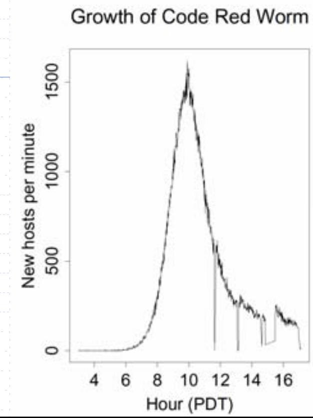
18

Code Red of July 13 and July 19

- Initial release of July 13
  - 1st through 20th month: Spread
    - via random scan of 32-bit IP addr space
  - 20th through end of each month: attack.
    - Flooding attack against 198.137.240.91 (*www.whitehouse.gov*)
  - Failure to seed random number generator ⇒ *linear growth*
- Revision released July 19, 2001.
  - White House responds to threat of flooding attack by changing the address of *www.whitehouse.gov*
  - Causes Code Red to die for date ≥ 20th of the month.
  - But: this time random number generator correctly seeded

Slides: Vern Paxson

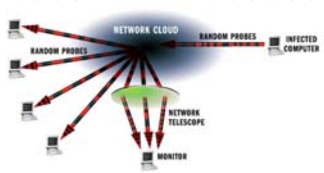
19



Slide: Vern Paxson

20

Measuring activity: network telescope



- Monitor cross-section of Internet address space, measure traffic
  - "Backscatter" from DOS floods
  - Attackers probing blindly
  - Random scanning from worms
- LBNL's cross-section: 1/32,768 of Internet
- UCSD, UWisc's cross-section: 1/256.

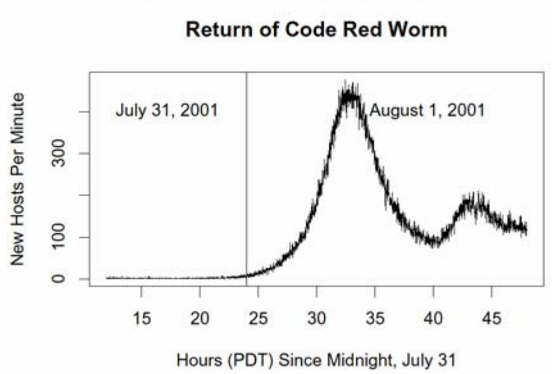
21

Spread of Code Red

- Network telescopes estimate of # infected hosts: 360K. (Beware DHCP & NAT)
- Course of infection fits classic *logistic*.
- Note: larger the vulnerable population, *faster* the worm spreads.
- That night (⇒ 20th), worm dies …
  … except for hosts with inaccurate clocks!
- It just takes one of these to restart the worm on August 1st …

Slides: Vern Paxson

22



Slides: Vern Paxson

23

Code Red 2

- Released August 4, 2001.
- Comment in code: "Code Red 2."
  - But in fact completely different code base.
- Payload: a root backdoor, resilient to reboots.
- Bug: crashes NT, only works on Windows 2000.
- *Localized scanning*: prefers nearby addresses.
- Kills Code Red 1.
- Safety valve: programmed to die Oct 1, 2001.

Slides: Vern Paxson

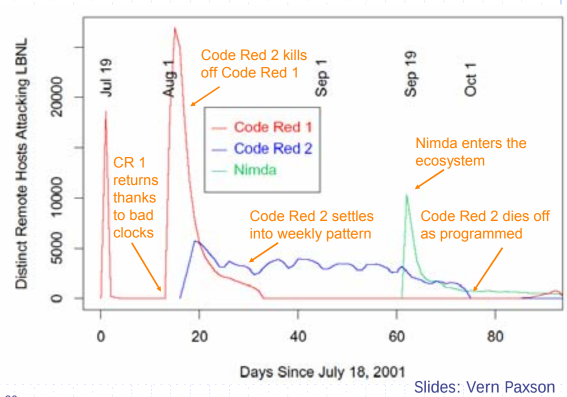
24

Striving for Greater Virulence: Nimda

- Released September 18, 2001.
- Multi-mode spreading:
  - attack IIS servers via infected clients
  - email itself to address book as a virus
  - copy itself across open network shares
  - modifying Web pages on infected servers w/ client exploit
  - scanning for Code Red II backdoors (!)
- ⇒ worms form an *ecosystem*!
- Leaped across firewalls.

Slides: Vern Paxson

25



Slides: Vern Paxson

26

Workshop on Rapid Malcode

- WORM '05
  - **Proc 2005 ACM workshop on Rapid malcode**
- WORM '04
  - **Proc 2004 ACM workshop on Rapid malcode**
- WORM '03
  - **Proc 2003 ACM workshop on Rapid malcode**

27

How do worms propagate?

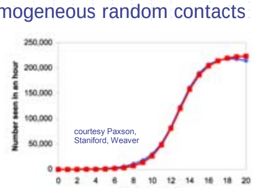
- Scanning worms
  - Worm chooses "random" address
- Coordinated scanning
  - Different worm instances scan different addresses
- Flash worms
  - Assemble tree of vulnerable hosts in advance, propagate along tree
    - Not observed in the wild, yet
    - Potential for 106 hosts in < 2 sec ! [Staniford]
- Meta-server worm
  - Ask server for hosts to infect (e.g., Google for "powered by phpbb")
- Topological worm:
  - Use information from infected hosts (web server logs, email address books, config files, SSH "known hosts")
- Contagion worm
  - Propagate parasitically along with normally initiated communication

28

How fast are scanning worms?

- Model propagation as infectious epidemic
  - Simplest version: Homogeneous random contacts

N: population size
S(t): susceptible hosts at time t
I(t): infected hosts at time t
ß: contact rate
i(t): I(t)/N, s(t): S(t)/N



$$\frac{dI}{dt} = \beta \frac{IS}{N}$$
$$\frac{dS}{dt} = -\beta \frac{IS}{N}$$
$$\frac{di}{dt} = \beta i(1-i)$$
$$i(t) = \frac{e^{\beta(t-T)}}{1+e^{\beta(t-T)}}$$

29

Shortcomings of simplified model

- Prediction is faster than observed propagation
- Possible reasons
  - Model ignores infection time, network delays
  - Ignores reduction in vulnerable hosts by patching
- Model supports unrealistic conclusions
  - Example: When the Top-100 ISP's deploy containment strategies, they still can not prevent a worm spreading at 100 probes/sec from affecting 18% of the internet, no matter what the reaction time of the system towards containment

30

Analytical Active Worm Propagation Model

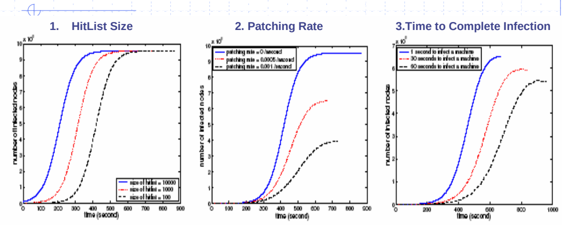
[Chen et al., Infocom 2003]

- More detailed discrete time model
  - Assume infection propagates in one time step
  - Notation
    - N – number of vulnerable machines
    - h – "hitlist: number of infected hosts at start
    - s – scanning rate: # of machines scanned per infection
    - d – death rate: infections detected and eliminated
    - p – patching rate: vulnerable machines become invulnerable
    - At time i, n_i are infected and m_i are vulnerable
- Discrete time difference equation
  - Guess random IP addr, so infection probability $(m_i-n_i)/2^{32}$
  - Number infected reduced by $pn_i + dn_i$

$$n_{i+1} = (1-d-p)n_i + [(1-p)^i N - n_i][1-(1-\frac{1}{2^{32}})^{sn_i}] \quad (1)$$

31

Effect of parameters on propagation



(Plots are for 1M vulnerable machines, 100 scans/sec, death rate 0.001/second

Other models:
Wang et al, *Modeling Timing Parameters …* , WORM '04 (includes delay)
Ganesh et al, *The Effect of Network Topology …*, Infocom 2005 (topology)

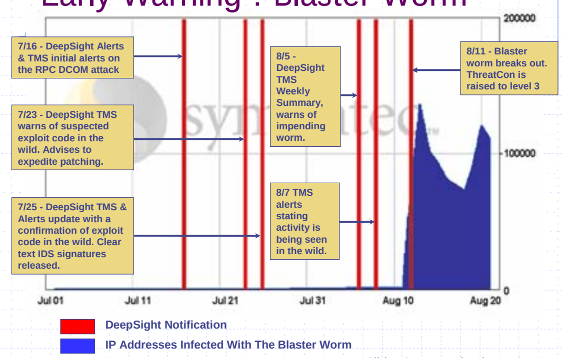
32

Worm Detection and Defense

- Detect via *honeyfarms*: collections of "honeypots" fed by a network telescope.
  - Any outbound connection from honeyfarm = worm. (at least, that's the theory)
  - Distill *signature* from inbound/outbound traffic.
  - If telescope covers N addresses, expect detection when worm has infected 1/N of population.
- Thwart via *scan suppressors*: network elements that block traffic from hosts that make failed connection attempts to too many other hosts

33

Early Warning : Blaster Worm

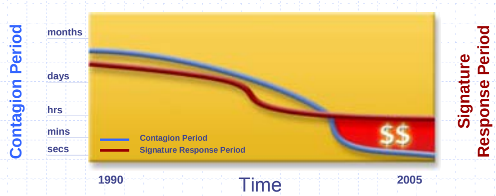


Slide: Carey Nachenberg, Symantec

34

Need for automation

- Current threats can spread faster than defenses can reaction
- Manual capture/analyze/signature/rollout model too slow



Slide: Carey Nachenberg, Symantec

35

Signature inference

- Challenge
  - need to automatically learn a content "signature" for each new worm – potentially in less than a second!
- Some proposed solutions
  - Singh et al, Automated Worm Fingerprinting, OSDI '04
  - Kim et al, Autograph: Toward Automated, Distributed Worm Signature Detection, USENIX Sec '04

36

Signature inference

- Monitor network and look for strings common to traffic with worm-like behavior
  - Signatures can then be used for content filtering

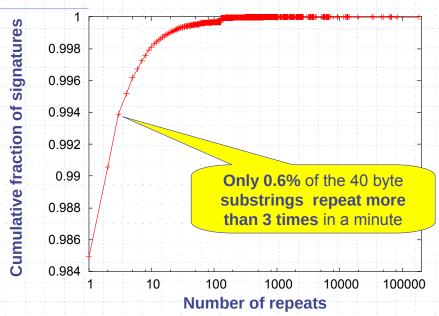
37 Slide: S Savage

Content sifting

- Assume there exists some (relatively) unique invariant bitstring *W* across all instances of a particular worm (*true today, not tomorrow...*)
- Two consequences
  - **Content Prevalence**: *W* will be more common in traffic than other bitstrings of the same length
  - **Address Dispersion**: the set of packets containing *W* will address a disproportionate number of distinct sources and destinations
- *Content sifting*: find *W*s with high content prevalence and high address dispersion and drop that traffic

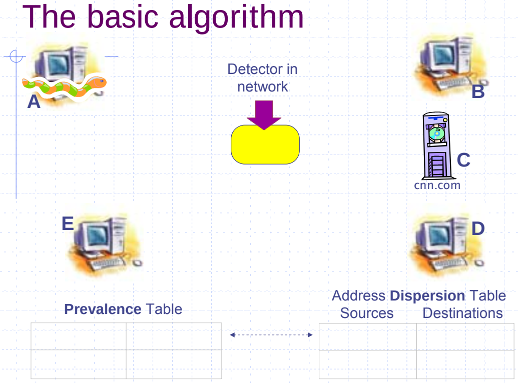
38 Slide: S Savage

Observation: High-prevalence strings are rare



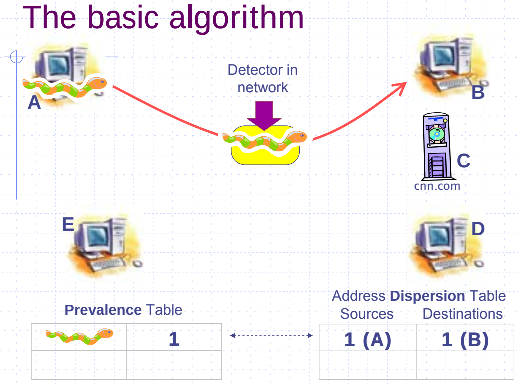
(Stefan Savage, UCSD *)

The basic algorithm



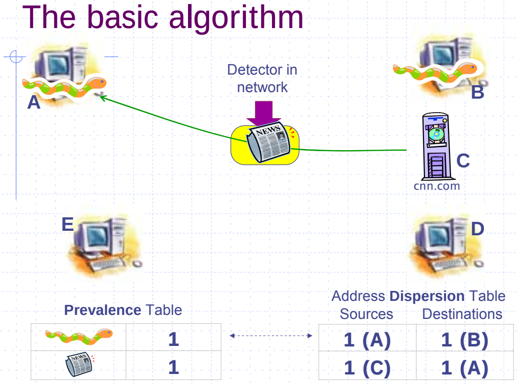
(Stefan Savage, UCSD *)

The basic algorithm



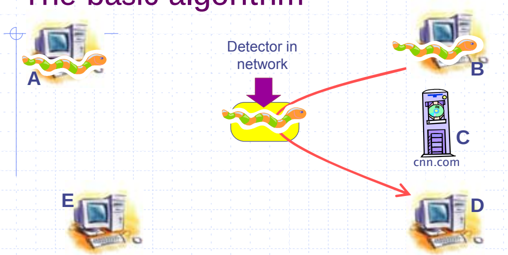
(Stefan Savage, UCSD *)

The basic algorithm



(Stefan Savage, UCSD *)

The basic algorithm



Prevalence Table

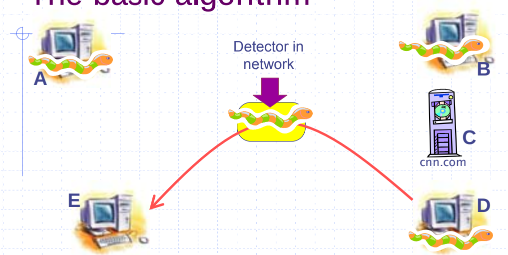
(worm)	2
(newspaper)	1

Address Dispersion Table

Sources	Destinations
2 (A,B)	2 (B,D)
1 (C)	1 (A)

(Stefan Savage, UCSD *)

The basic algorithm



Prevalence Table

(worm)	3
(newspaper)	1

Address Dispersion Table

Sources	Destinations
3 (A,B,D)	3 (B,D,E)
1 (C)	1 (A)

(Stefan Savage, UCSD *)

Challenges

- Computation
  - To support a 1Gbps line rate we have 12us to process each packet, at 10Gbps 1.2us, at 40Gbps…
    - Dominated by memory references; state expensive
  - Content sifting requires looking at every byte in a packet
- State
  - On a fully-loaded 1Gbps link a naïve implementation can easily consume 100MB/sec for table
  - Computation/memory duality: on high-speed (ASIC) implementation, latency requirements may limit state to on-chip SRAM

(Stefan Savage, UCSD *)

Which substrings to index?

- **Approach 1: Index all substrings**
  - Way too many substrings → too much computation → too much state
- **Approach 2: Index whole packet**
  - Very fast but trivially evadable (e.g., Witty, Email Viruses)
- **Approach 3: Index all contiguous substrings of a fixed length 'S'**
  - Can capture all signatures of length 'S' and larger

A B C D E F G H I J K

(Stefan Savage, UCSD *)

How to represent substrings?

- Store **hash** instead of literal to reduce state
- **Incremental hash** to reduce computation
- **Rabin fingerprint** is one such efficient incremental hash function [Rabin81,Manber94]
  - One multiplication, addition and mask per byte

P1 R A N D A B C D O M

Fingerprint = 11000000

P2 R A B C D A N D O M

Fingerprint = 11000000

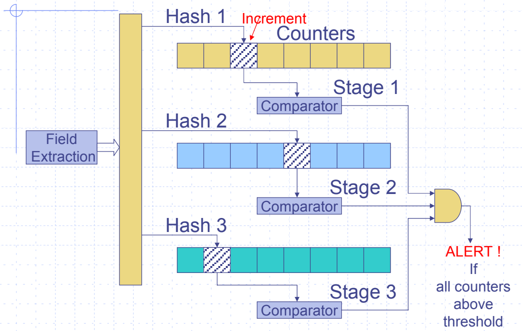
(Stefan Savage, UCSD *)

How to subsample?

- **Approach 1: sample packets**
  - If we chose 1 in N, detection will be slowed by N
- **Approach 2: sample at particular byte offsets**
  - Susceptible to simple evasion attacks
  - No guarantee that we will sample same sub-string in every packet
- **Approach 3: sample based on the hash of the substring**

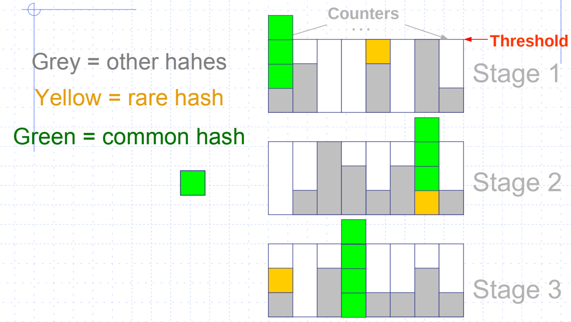
(Stefan Savage, UCSD *)

Finding "heavy hitters" via Multistage Filters



(Stefan Savage, UCSD *)

Multistage filters in action



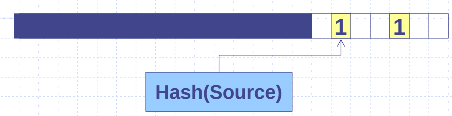
(Stefan Savage, UCSD *)

Observation:
High *address dispersion* is rare too

- Naïve implementation might maintain a list of sources (or destinations) for each string hash
- But dispersion **only** matters if its *over* threshold
  - Approximate counting may suffice
  - **Trades accuracy for state in data structure**
- **Scalable Bitmap Counters**
  - Similar to multi-resolution bitmaps [Estan03]
  - Reduce memory by 5x for modest accuracy error

(Stefan Savage, UCSD *)

Scalable Bitmap Counters



- Hash : based on Source (or Destination)
- Sample : keep only a sample of the bitmap
- Estimate : scale up sampled count
- Adapt : periodically increase scaling factor

Error Factor = $2/(2^{numBitmaps}-1)$

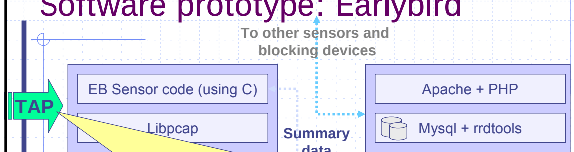
(Stefan Savage, UCSD *)

Content sifting summary

- Index fixed-length substrings using incremental hashes
- Subsample hashes as function of hash value
- Multi-stage filters to filter out uncommon strings
- Scalable bitmaps to tell if number of distinct addresses per hash crosses threshold
- This is fast enough to implement

(Stefan Savage, UCSD *)

Software prototype: Earlybird

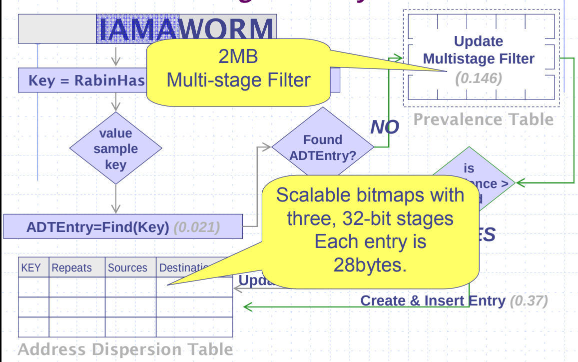


Setup 1: Large fraction of the UCSD campus traffic,
Traffic mix: approximately 5000 end-hosts, dedicated servers for campus wide services (DNS, Email, NFS etc.)
Line-rate of traffic varies between 100 & 500Mbps.

Setup 2: Fraction of local ISP Traffic,
Traffic mix: dialup customers, leased-line customers
Line-rate of traffic is roughly 100Mbps.

(Stefan Savage, UCSD *)

Content Sifting in Earlybird



(Stefan Savage, UCSD *)

Content sifting overhead

- Mean per-byte processing cost
  - **0.409 microseconds**, without value sampling
  - **0.042 microseconds**, with 1/64 value sampling (~60 microseconds for a 1500 byte packet, can keep up with 200Mbps)
- Additional overhead in per-byte processing cost for flow-state maintenance (if enabled):
  - **0.042 microseconds**

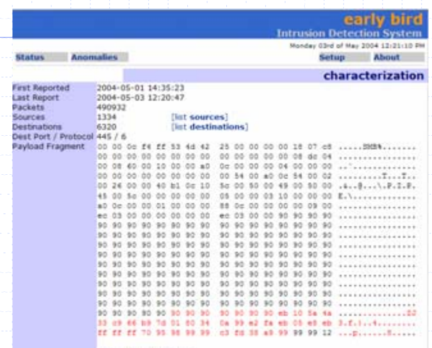
(Stefan Savage, UCSD *)

Experience

- Quite good.
  - Detected and automatically generated signatures for every known worm outbreak over eight months
  - Can produce a precise signature for a new worm in a fraction of a second
  - Software implementation keeps up with 200Mbps
- Known worms detected:
  - Code Red, Nimda, WebDav, Slammer, Opaserv, …
- Unknown worms (with no public signatures) detected:
  - MsBlaster, Bagle, Sasser, Kibvu, …

(Stefan Savage, UCSD *)

Sasser



(Stefan Savage, UCSD *)

False Negatives

- Easy to prove presence, impossible to prove absence
- **Live evaluation**: over 8 months detected every worm outbreak reported on popular security mailing lists
- **Offline evaluation**: several traffic traces run against both Earlybird and Snort IDS (w/all worm-related signatures)
  - Worms not detected by Snort, but detected by Earlybird
  - The converse never true

(Stefan Savage, UCSD *)

False Positives

- **Common protocol headers**
  - Mainly HTTP and SMTP headers
  - Distributed (P2P) system protocol headers
  - **Procedural whitelist**
    - Small number of popular protocols
- **Non-worm epidemic Activity**
  - **SPAM**
  - BitTorrent

```
GNUTELLA.CONNECT
/0.6..X-Max-TTL:
.3..X-Dynamic-Qu
erying:.0.1..X-V
ersion:.4.0.4..X
-Query-Routing:.
0.1..User-Agent:
.LimeWire/4.0.6.
.Vendor-Message:
.0.1..X-Ultrapee
r-Query-Routing:
```

(Stefan Savage, UCSD *)

TaintCheck Worm Detection

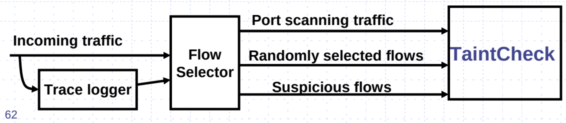
Song et al.

- Previous work look for “worm-like” behavior
  - Port-scanning [Autograph], contacting honey pots [Honeycomb], traffic patterns [Earlybird]
  - False negatives: Non-scanning worms
  - False positives: Easy for attackers to raise false alarms
- TaintCheck approach: cause-based detection
  - Use distributed TaintCheck-protected servers
  - Watch behavior of host after worm arrives
  - Can be effective for nonscanning or polymorphic worms
  - Difficult for attackers to raise false alarms

61

Fast, Low-Cost Distributed Detection

- Low load servers & Honeypots:
  - Monitor all incoming requests
  - Monitor port scanning traffic
- High load servers:
  - Randomly select requests to monitor
  - Select suspicious requests to monitor
    - When server is abnormal
      - E.g., server becomes client, server starts strange network/OS activity
    - Anomalous requests



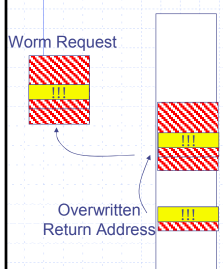
62

TaintCheck Approach

- Observation:
  - certain parts in packets need to stay invariant even for polymorphic worms
- Automatically identify invariants in packets for signatures
  - More sophisticated signature types
  - Semantic-based signature generation
- Advantages
  - Fast
  - Accurate
  - Effective against polymorphic worms

63

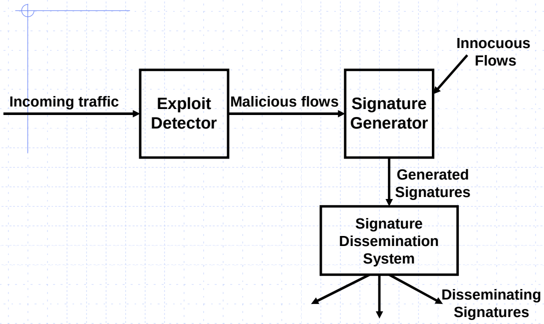
Semantic-based Signature Generation (I)



- Identifying invariants using semantic-based analysis
- Example invariants (I):
  - Identify overwrite value
  - Trace back to value in original request
- Experiment: ATPHttpd exploit
  - Identified overwrite return address
  - Used top 3 bytes as signature
  - Signature had 1 false positive out of 59,280 HTTP requests

64

Sting Architecture



65

Sting Evaluation

- Slammer worm attack:
  - 100,000 vulnerable hosts
  - 4000 scans per second
  - Effective contact rate r: 0.1 per second
- Sting evaluation I:
  - 10% deployment, 10% sample rate
  - Dissemination rate: 2*r = 0.2 per second
  - Fraction of protected vulnerable host: 70%
- Sting evaluation II:
  - 1% deployment, 10% sample rate
  - 10% vulnerable host protected for dissemination rate 0.2 per second
  - 98% vulnerable host protected for dissemination rate 1 per second

66

Generic Exploit Blocking

- Idea
  - Write a network IPS signature to generically detect and block all future attacks on a vulnerability
  - Different from writing a signature for a specific exploit!
- Step #1: Characterize the vulnerability "shape"
  - Identify fields, services or protocol states that must be present in attack traffic to exploit the vulnerability
  - Identify data footprint size required to exploit the vulnerability
  - Identify locality of data footprint; will it be localized or spread across the flow?
- Step #2: Write a generic signature that can detect data that "mates" with the vulnerability shape
- Similar to Shield research from Microsoft

Generic Exploit Blocking Example #1

Consider MS02-039 Vulnerability (SQL Buffer Overflow):

Field/service/protocol
UDP port 1434
Packet type: 4

Minimum data footprint
Packet size > 60 bytes

Data Localization
Limited to a single packet

```
BEGIN
 DESCRIPTION: MS02-039
 NAME: MS SQL Vuln
 TRANSIT-TYPE: UDP
 TRIGGER: ANY:ANY->ANY:1434
 OFFSET: 0, PACKET
 SIG-BEGIN
   "\x04<getpacketsize(r0)>
   <inrange(r0,61,1000000)>
   <reportid()>"
 SIG-END
END
```


Generic Exploit Blocking Example #2

Consider MS03-026 Vulnerability (RPC Buffer Overflow):

Field/service/protocol
RPC request on TCP/UDP 135
szName field in CoGetInstanceFromFile func.

Minimum data footprint
Arguments > 62 bytes

Data Localization
Limited to 256 bytes from start of RPC bind command

```
BEGIN
 DESCRIPTION: MS03-026
 NAME: RPC Vulnerability
 TRANSIT-TYPE: TCP, UDP
 TRIGGER: ANY:ANY->ANY:135
 SIG-BEGIN
   "\x05\x00\x0B\x03\x10\x00\x00
    (about 50 more bytes...)
    \x00\x00.*\x05\x00
    <forward(5)><getbeword(r0)>
    <inrange(r0,63,20000)>
    <reportid()>"
 SIG-END
END
```


Conclusions

- Worm attacks
  - Many ways for worms to propagate
  - Propagation time is increasing
  - Polymorphic worms, other barriers to detection
- Detect
  - Traffic patterns: EarlyBird
  - Watch attack: TaintCheck and Sting
  - Look at vulnerabilities: Generic Exploit Blocking
- Disable
  - Generate worm signatures and use in network or host-based filters