



Security in knowledge

UNDERSTANDING AND FIGHTING EVASIVE MALWARE

Christopher Kruegel

Lastline Inc. and UC Santa Barbara

RSACONFERENCE
EUROPE 2013

Session ID: HTA-W10

Session Classification: Advanced

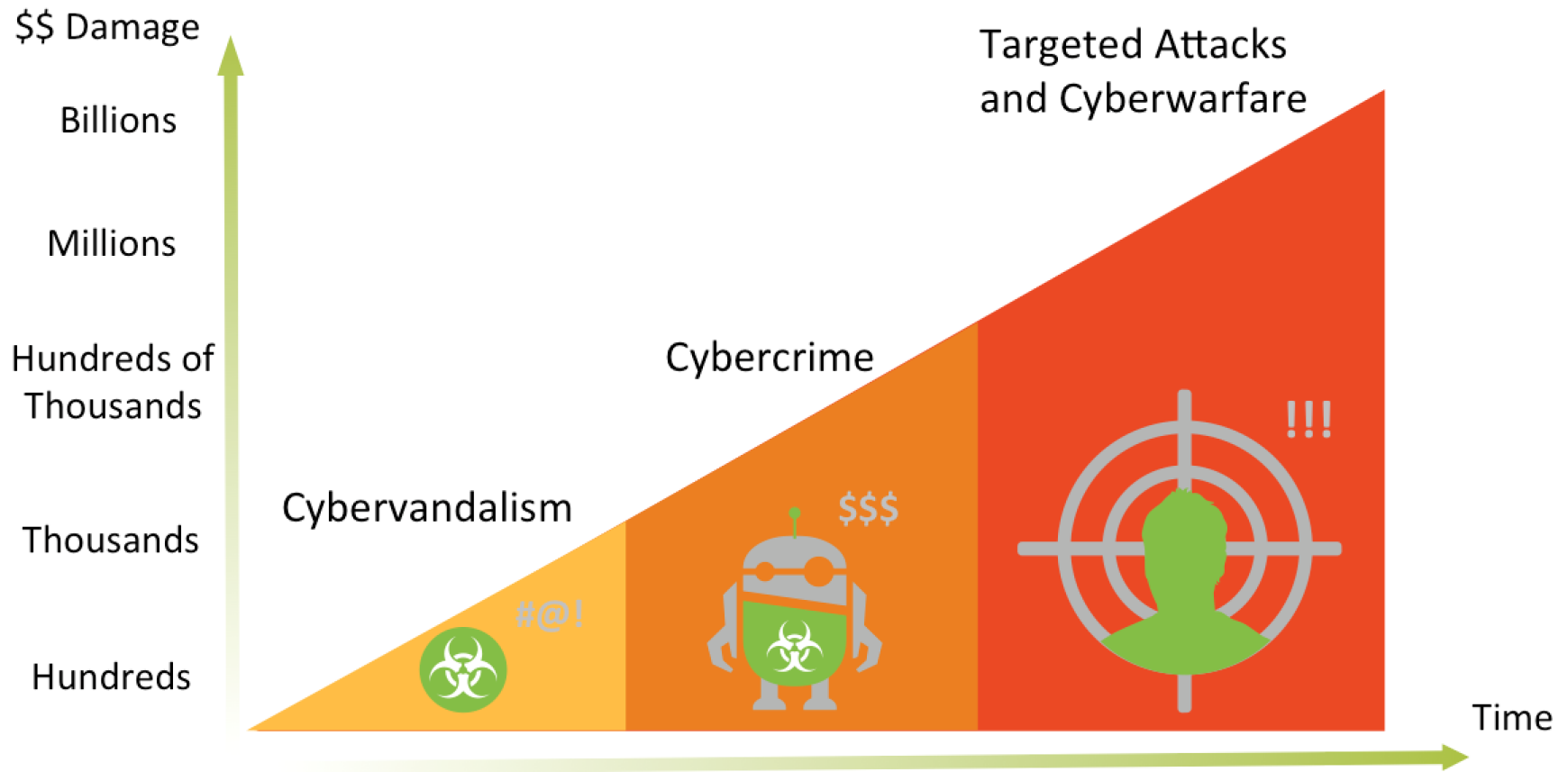
Who am I?

- ▶ Professor in Computer Science at UC Santa Barbara
 - ▶ 100+ systems security papers in academic conferences
 - ▶ started malware research in about 2004
 - ▶ built and released practical systems (Anubis, Wepawet, ...)
- ▶ Co-founder and Chief Scientist at Lastline, Inc.
 - ▶ Lastline offers protection against zero-day threats and advanced malware
 - ▶ venue to commercialize our academic research

What are we talking about?

- ▶ Evolution of malicious code and automated malware analysis
- ▶ Evasion as a significant threat to automated analysis
 - ▶ detect analysis environment
 - ▶ detect analysis system
 - ▶ avoid being seen by automated analysis
- ▶ Improvements to analysis systems
 - ▶ automate defenses against common evasion approaches

Evolution of Malware



Malware Analysis

The screenshot displays the OllyDbg interface for the process 601e77d9.exe. The CPU window shows assembly instructions for the main thread in the ntdll module. The registers window shows the current state of the CPU registers, with EAX containing 00000018. The threads window shows a single thread with ID 0000077C. The call stack window shows the current function being executed, <JMP.&MSUBUM60.#100>, and its callers.

CPU - main thread, module ntdll

Address	Hex dump	ASCII	Comment
7C90E8A6	68 00E9907C		PUSH ntdll.7C90E900
7C90E8B0	64:A1 00000000		MOV EAX, DWORD PTR FS:[0]
7C90E8B6	50		PUSH EAX
7C90E8B7	8B4424 10		MOV EAX, DWORD PTR SS:[ESP+10]
7C90E8B8	896C24 10		MOV DWORD PTR SS:[ESP+10], EBP
7C90E8BF	8D6C24 10		LEA EBP, DWORD PTR SS:[ESP+10]
7C90E8C3	2BE0		SUB ESP, EAX
7C90E8C5	53		PUSH EBX
7C90E8C6	56		PUSH ESI
7C90E8C7	57		PUSH EDI
7C90E8C8	8B45 F8		MOV EAX, DWORD PTR SS:[EBP-8]
7C90E8CB	8965 E8		MOV DWORD PTR SS:[EBP-18], ESP
7C90E8CE	50		PUSH EAX
7C90E8CF	8B45 FC		MOV EAX, DWORD PTR SS:[EBP-4]
7C90E8D2	C745 FC FFFFFFFF		MOV DWORD PTR SS:[EBP-4], -1
7C90E8D9	8945 F8		MOV DWORD PTR SS:[EBP-8], EAX
7C90E8DC	8D45 F0		LEA EAX, DWORD PTR SS:[EBP-10]
7C90E8DF	64:A3 00000000		MOV DWORD PTR FS:[0], EAX
7C90E8E5	C3		RETN
7C90E8E6	8B4D F0		MOV ECX, DWORD PTR SS:[EBP-10]

Registers (FPU)

Register	Value
EAX	00000018
ECX	0012FFB0
EDX	7C90E4F4 ntdll.KiFastSysT
EBX	7FFD4000
ESP	0012FEF0
EBP	0012FF3C
ESI	00020000
EDI	7C910208 ntdll.7C910208
EIP	7C90E8BB ntdll.7C90E8BB

Threads

Ident	Entry	Data block	Last error
0000077C	004010B8	7FFDF000	ERROR_SUCCESS

Call stack of main thread

Address	Stack	Procedure / arguments	Called from
0012FEF8	7C9103F9	? ntdll.7C90E8AB	ntdll.7C9103F4
0012FF04	7C801F10	? ntdll.RtlAcquirePebLock	kerne!32.7C801F0A
0012FF40	734235E3	? kernel32.GetStartupInfoA	MSUBUM60.734235D0
0012FF44	0012FF58	pStartupInfo = 0012FF58	
0012FFBC	004010C2	? <JMP.&MSUBUM60.#100>	601e77d9.004010BD

Malware Analysis

The screenshot displays the OllyDbg interface for the process 601e77d9.exe. The CPU window shows assembly instructions for the main thread in module ntdll. The registers window shows the EAX register containing 00000018. The threads window shows a single thread with ID 0000077C. The Windows Task Manager window is open, showing a list of running processes.

CPU - main thread, module ntdll

Address	Hex	Disassembly
7C90E8A8	68 00E9907C	PUSH ntdll.7C90E900
7C90E8B0	64:A1 00000000	MOV EAX, DWORD PTR FS:[0]
7C90E8B6	50	PUSH EAX
7C90E8B7	8B4424 10	MOV EAX, DWORD PTR SS:[ESP+10]
7C90E8B8	896C24 10	MOV DWORD PTR SS:[ESP+10], EBP
7C90E8BF	8D6C24 10	LEA EBP, DWORD PTR SS:[ESP+10]
7C90E8C3	2BE0	SUB ESP, EAX
7C90E8C5	53	PUSH EBX
7C90E8C6	56	PUSH ESI
7C90E8C7	57	PUSH EDI
7C90E8C8	8B45 F8	MOV EAX, DWORD PTR SS:[EBP-8]
7C90E8CB	8965 E8	MOV DWORD PTR SS:[EBP-18], ESP
7C90E8CE	50	PUSH EAX
7C90E8CF	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]
7C90E8D2	C745 FC FFFFFFFF	MOV DWORD PTR SS:[EBP-4], -1
7C90E8D9	8945 F8	MOV DWORD PTR SS:[EBP-8], EAX
7C90E8DC	8D45 F0	LEA EAX, DWORD PTR SS:[EBP-10]
7C90E8DF	64:A3 00000000	MOV DWORD PTR FS:[0], EAX
7C90E8E5	C3	RETN
7C90E8E6	8B4D F0	MOV ECX, DWORD PTR SS:[EBP-10]

Registers (FPU)

Register	Value
EAX	00000018
ECX	0012FFFE
EDX	7C90E4F0
EBX	7FFD4000
ESP	0012FFFE
EBP	0012FFFE
ESI	00020000
EDI	7C910200
EIP	7C90E8E6

Threads

Ident	Entry
0000077C	004010B8

Windows Task Manager

Image Name	User Name	CPU	Mem Usage
wuauclt.exe	SYSTEM	00	2,420 K
wscntfy.exe	user	00	680 K
wpabaln.exe	user	00	2,784 K
winlogon.exe	SYSTEM	00	1,732 K
lurdxvc.exe	user	00	388 K
taskmgr.exe	user	02	4,296 K
System Idle Process	SYSTEM	98	16 K
System	SYSTEM	00	36 K
svchost.exe	LOCAL SERVICE	00	740 K
svchost.exe	NETWORK SERVICE	00	1,340 K
svchost.exe	SYSTEM	00	8,176 K
svchost.exe	NETWORK SERVICE	00	1,628 K
svchost.exe	SYSTEM	00	1,308 K
spoolsv.exe	SYSTEM	00	1,488 K
smss.exe	SYSTEM	00	56 K
services.exe	SYSTEM	00	1,376 K
OLLYDBG.EXE	user	00	7,588 K
lsass.exe	SYSTEM	00	968 K
jusched.exe	user	00	520 K
explorer.exe	user	00	13,452 K

Stack of main thread

Address	Stack	Procedure / arguments	Called from
0012FEF8	7C9103F9	? ntdll.7C90E8AB	ntdll.7C9103F4
0012FF04	7C801F10	? ntdll.RtlAcquirePebLock	kernel32.7C801F0A
0012FF40	734235E3	? kernel32.GetStartupInfoA	MSUBUM60.734235D0
0012FF44	0012FF58	pStartupInfo = 0012FF58	
0012FFBC	004010C2	? <JMP.&MSUBUM60.#100>	601e77d9.004010BD

Malware Analysis

The screenshot displays the OllyDbg interface for the process 601e77d9.exe. The CPU window shows assembly instructions for the main thread in module ntdll, including PUSH, MOV, LEA, SUB, and PUSH instructions. The Registers (FPU) window shows the current state of registers, with EAX at 00000018 and EIP at 7C90E0E. The Stack window shows the current stack frame with EBP at 0012F3C and SS at 0012F1F. The Network window shows a list of network traffic, including a GET request for /images/led/hp.php. The Windows Task Manager window is also visible, showing a list of running processes with columns for Image Name, User Name, CPU, and Mem Usage.

Image Name	User Name	CPU	Mem Usage
wuauclt.exe	SYSTEM	00	2,420 K
wscntfy.exe	user	00	680 K
wpabaln.exe	user	00	2,784 K
winlogon.exe	SYSTEM	00	1,732 K
...
gr.exe	user	02	4,296 K
n Idle Process	SYSTEM	98	16 K
n	SYSTEM	00	36 K
st.exe	LOCAL SERVICE	00	740 K
st.exe	NETWORK SERVICE	00	1,340 K
st.exe	SYSTEM	00	8,176 K
st.exe	NETWORK SERVICE	00	1,628 K
st.exe	SYSTEM	00	1,308 K
iv.exe	SYSTEM	00	1,488 K
exe	SYSTEM	00	56 K
es.exe	SYSTEM	00	1,376 K
bg.exe	user	00	7,588 K
exe	SYSTEM	00	968 K
sd.exe	user	00	520 K
er.exe	user	00	13,452 K

Malware Analysis

The screenshot illustrates a malware analysis environment using OllyDbg. The main window shows assembly code with registers (EAX, ECX, EDX, EBX, ESP, EBP, EIP) and their values. A Windows Task Manager window is overlaid, showing a list of running processes with columns for Image Name, User Name, CPU, and Mem Usage. A network traffic capture window is also visible, showing details for an HTTP GET request to /images/led/hp.php.

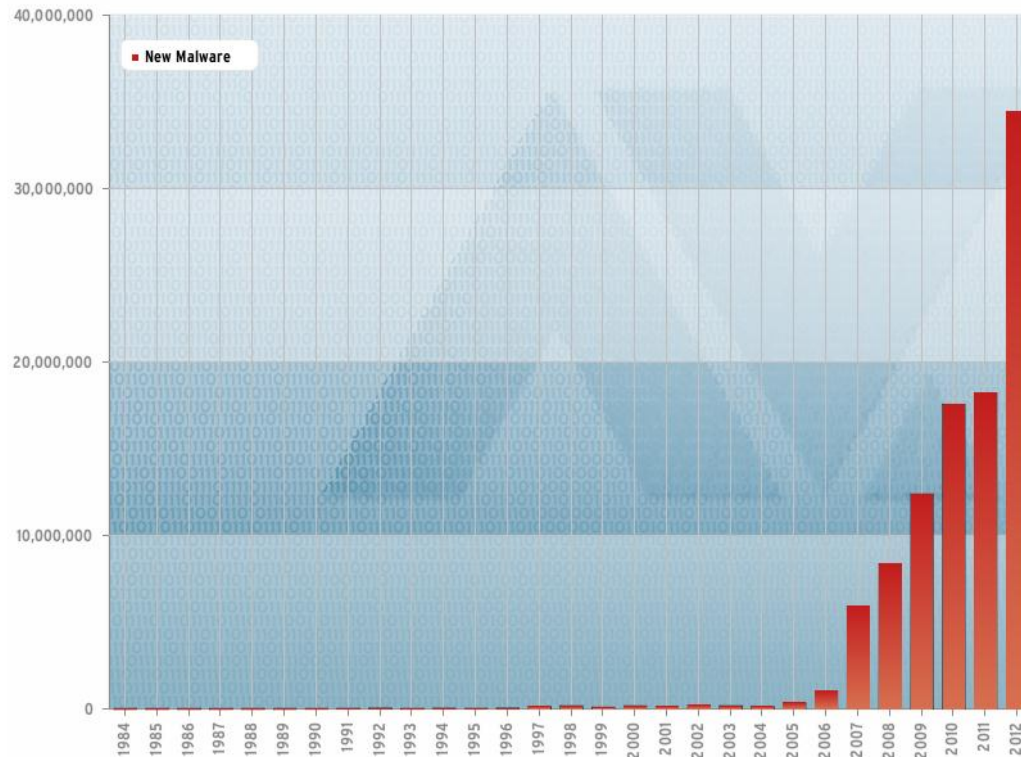
Image Name	User Name	CPU	Mem Usage
wuauclt.exe	SYSTEM	00	2,420 K
wscntfy.exe	user	00	680 K
wpabaln.exe	user	00	2,784 K
winlogon.exe	SYSTEM	00	1,732 K
...
gr.exe	user	02	4,296 K
n Idle Process	SYSTEM	98	16 K
n	SYSTEM	00	36 K
st.exe	LOCAL SERVICE	00	740 K
st.exe	NETWORK SERVICE	00	1,340 K
st.exe	SYSTEM	00	8,176 K
st.exe	NETWORK SERVICE	00	1,628 K
st.exe	SYSTEM	00	1,308 K
sv.exe	SYSTEM	00	1,488 K
exe	SYSTEM	00	56 K
es.exe	SYSTEM	00	1,376 K
DBG.EXE	user	00	7,588 K
exe	SYSTEM	00	968 K
sd.exe	user	00	520 K
rer.exe	user	00	13,452 K

Protocol	Info
TCP	netarx > http [SYN] Seq=0 Win=16384 Len=0 MSS=1460
TCP	http > netarx [SYN, ACK] Seq=80 Ack=1 Win=16384 Len=0
TCP	netarx > http [ACK] Seq=1 Ack=1 Win=16568 Len=0
HTTP	GET /images/led/hp.php HTTP/1.1
TCP	[TCP segment of a reassembled PDU]
TCP	netarx > http [ACK] Seq=197 Ack=1566 Win=16568 Len=0
TCP	[TCP segment of a reassembled PDU]
HTTP	HTTP/1.1 404 Unbeschrieben (text/html)
TCP	netarx > http [ACK] Seq=197 Ack=1893 Win=16234 Len=0
TCP	netarx > http [RST, ACK] Seq=197 Ack=1893 Win=0 Len=0

There is a lot of malware out there ...

New Malware

▶ All years ▶ Last 10 years ▶ Last 5 years ▶ Last 24 months ▶ Last 12 months



Last update: 03-16-2013 09:03

Copyright © AV-TEST GmbH, www.av-test.org

Automated Malware Analysis

- ▶ Aka sandbox
- ▶ Automation is great!
 - ▶ analysts do not need to look at each sample by hand (debugger)
 - ▶ only way to stem flood of samples and get scalability
 - ▶ can handle zero day threats (signature less defense)
- ▶ Implemented as instrumented execution environment
 - ▶ run program and observe its activity
 - ▶ make determination whether code is malicious or not

Automated Malware Analysis

- ▶ Not all sandboxes are equal!

It is easy to build a sandbox,
it is hard to build an effective sandbox!

Lawrence Orans
“The Executive's Guide to Cyberthreats”
(Gartner Symposium, October 2013)

Automated Malware Analysis

- ▶ Ask your vendor questions about their sandbox
 - ▶ what files are supported (executables, documents, more ...)
 - ▶ how effective is classification of malicious behaviors
 - ▶ how effective is sandbox in eliciting behaviors (evasion!)

Automated Malware Analysis

- ▶ Anubis: ANalyzing Unknown Binaries
(dynamic malware analysis environment)



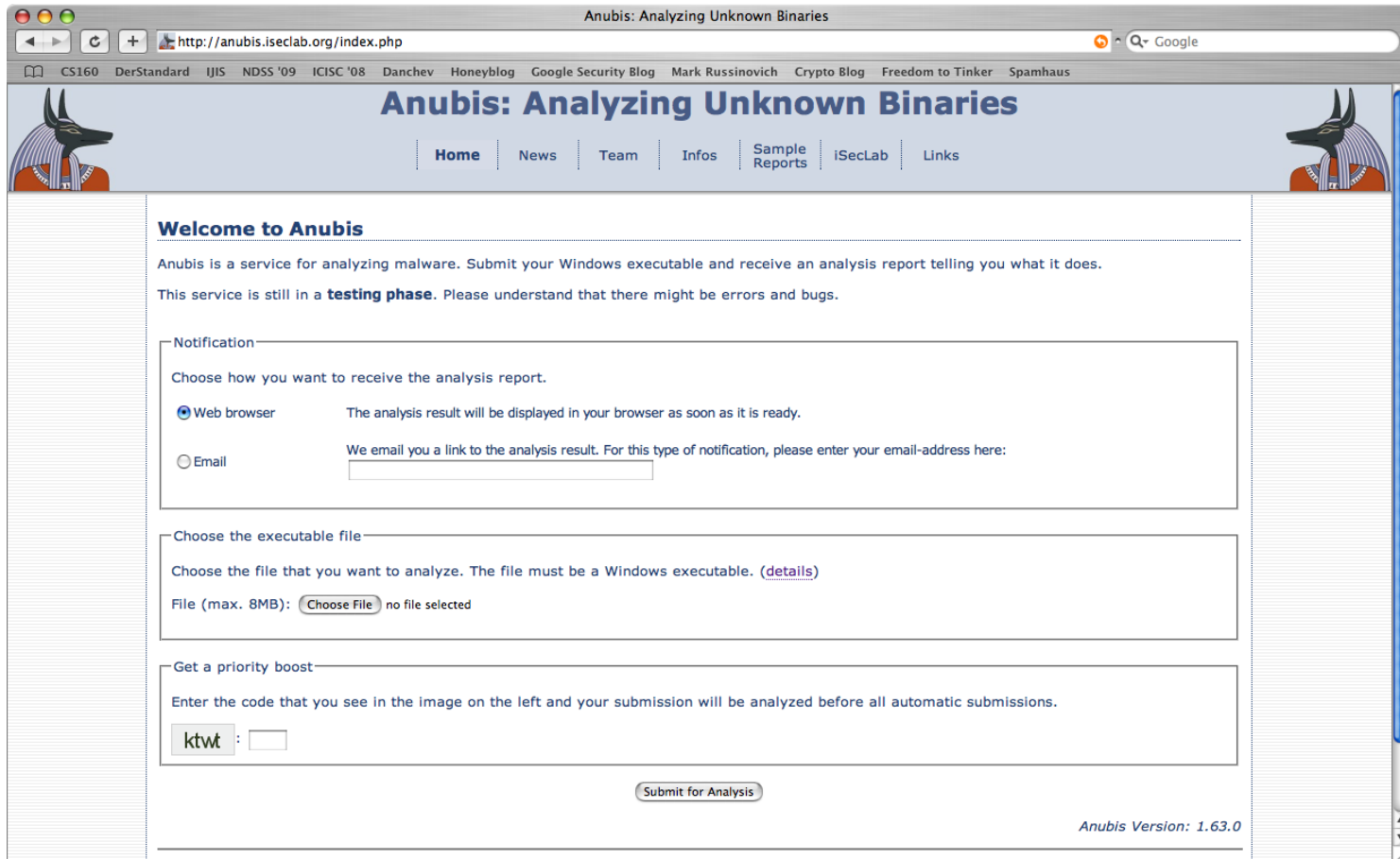
Automated Malware Analysis

▶ Anubis: ANalyzing Unknown Binaries (dynamic malware analysis environment)



- ▶ based on system/CPU emulator (Qemu)
- ▶ can see every instruction!
- ▶ monitors system activity from the outside (stealthier)
- ▶ requires mechanisms to handle semantic gap
- ▶ general platform on which additional components can be built
- ▶ supports dynamic data flow analysis (taint tracking)

Automated Malware Analysis



The screenshot shows a web browser window with the URL `http://anubis.iseclab.org/index.php`. The page title is "Anubis: Analyzing Unknown Binaries". The navigation menu includes "Home", "News", "Team", "Infos", "Sample Reports", "iSecLab", and "Links". The main content area is titled "Welcome to Anubis" and contains the following text: "Anubis is a service for analyzing malware. Submit your Windows executable and receive an analysis report telling you what it does. This service is still in a **testing phase**. Please understand that there might be errors and bugs."

The form includes a "Notification" section with two radio buttons: "Web browser" (selected) and "Email". The "Email" option has a text input field for an email address. Below this is a "Choose the executable file" section with a "Choose File" button and the text "no file selected". The "Get a priority boost" section has a CAPTCHA image showing the text "ktwt" and an input field for the user's response. A "Submit for Analysis" button is located at the bottom of the form. The version number "Anubis Version: 1.63.0" is displayed in the bottom right corner.

VM Engine versus CPU Emulation



callq 0x100070478 ; symbol stub for: _open



callq 0x1000704b4 ; symbol stub for: _read



callq 0x1000702b6 ; symbol stub for: _close



```
cmpl    $0x0c,%ebx
je      0x10000f21e
xorl    %esi,%esi
movq    %r15,%rdi
xorl    %eax,%eax
callq   0x100070478 ; symbol stub for: _open
movl    %eax,%r12d
testl   %eax,%eax
js      0x10000f21e
leaq    0xffffffff70(%rbp),%rcx
movq    %rcx,0xfffffec0(%rbp)
movl    $0x00000050,%edx
movq    %rcx,%rsi
movl    %eax,%edi
callq   0x1000704b4 ; symbol stub for: _read
movq    %rax,%r13
movl    %eax,%r14d
movl    %r12d,%edi
callq   0x1000702b6 ; symbol stub for: _close
cmpl    $0x02,%r13d
jle     0x10000f21e
```


Dynamic Data Flow Analysis

▶ Data tainting

- ▶ if any byte of any input value is tainted, then all bytes of the output are tainted
(e.g., `add %eax, %ebx`)

▶ Address tainting

- ▶ in addition, if any byte of any input value that is involved in the address computation of a source memory operand is tainted, then the output is tainted
(e.g., `mov %eax, (%ecx, %ebx, 2)`)

Evasions



Security in knowledge



RSAC CONFERENCE
EUROPE 2013

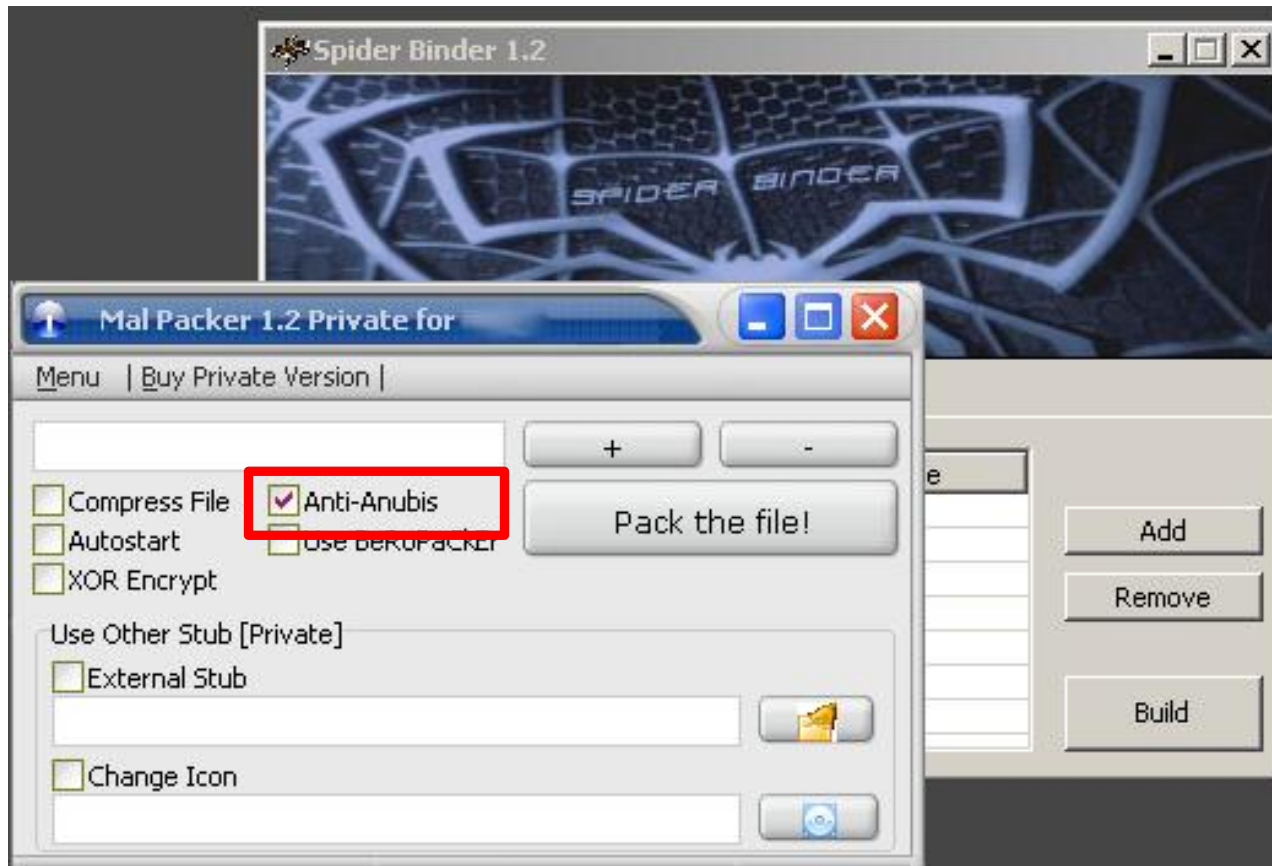
Evasion

- ▶ Malware authors are not stupid
 - ▶ they got the news that sandboxes are all the rage now
 - ▶ since the code is executed, malware authors have options ..
- ▶ Evasion
 - ▶ develop code that exhibits no malicious behavior in sandbox, but that infects the intended target
 - ▶ can be achieved in various ways

Evasion

- ▶ Malware can detect underlying runtime environment
 - ▶ differences between virtualized and bare metal environment
 - ▶ checks based on system (CPU) features
 - ▶ artifacts in the operating system
- ▶ Malware can detect signs of specific analysis environments
 - ▶ checks based on operating system artifacts (files, processes, ...)
- ▶ Malware can avoid being analyzed
 - ▶ tricks in making code run that analysis system does not see
 - ▶ wait until someone clicks something
 - ▶ time out analysis before any interesting behaviors are revealed
 - ▶ simple sleeps, but more sophisticated implementations possible

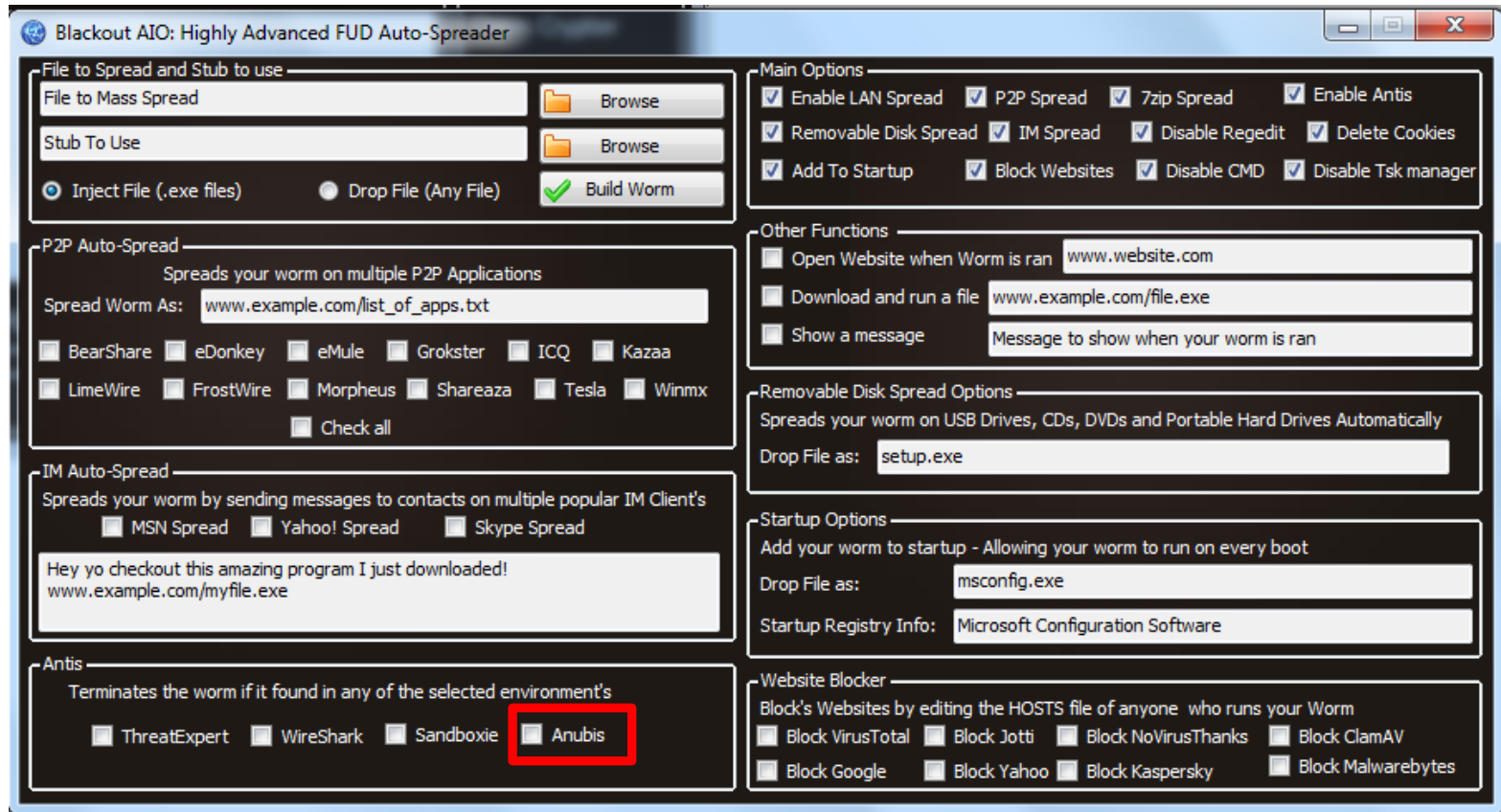
Evasion



Evasion



Evasion



Detect Runtime Environment

- ▶ Insufficient support from hardware for virtualization
 - ▶ J. Robin and C. Irvine: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor; Usenix Security Symposium, 2000
 - ▶ famous RedPill code snippet

Joanna Rutkowska

Swallowing the **Red Pill** is more or less equivalent to the following code (returns non zero when in Matrix):

```
int swallow_redpill () {
    unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
    *((unsigned*)&rpill[3]) = (unsigned)m;
    ((void(*)())&rpill)();
    return (m[5]>0xd0) ? 1 : 0;
}
```


Detect Runtime Environment

- ▶ Insufficient support from hardware for virtualization
 - ▶ J. Robin and C. Irvine: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor; Usenix Security Symposium, 2000
 - ▶ famous RedPill code snippet
- ▶ hardware assisted virtualization (Intel-VT and AMD-V) helps
- ▶ but systems can still be detected due to timing differences

Detect Runtime Environment

- ▶ CPU bugs or unfaithful emulation
 - ▶ invalid opcode exception, incorrect debug exception, ...
 - ▶ later automated in: R. Paleari, L. Martignoni, G. Roglia, D. Bruschi: A fistful of red-pills: How to automatically generate procedures to detect CPU emulators; Usenix Workshop on Offensive Technologies (WOOT), 2009
 - ▶ recently, we have seen malware make use of (obscure) math instructions
- ▶ The question is ... can malware really assume that a generic virtual machine implies an automated malware analysis system?

Detect Analysis Engine

- ▶ Check Windows XP Product ID

`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProductID`

- ▶ Check for specific user name, process names, hard disk names

`HKLM\SYSTEM\CURRENTCONTROLSET\SERVICES\DISK\ENUM`

- ▶ Check for unexpected loaded DLLs or Mutex names

- ▶ Check for color of background pixel

- ▶ Check of presence of 3-button mouse, keyboard layout, ...

Detect Analysis Engine

```
.text:00401E37  
.text:00401E39 loc_401E39: ; CODE XREF: .text:00401DCC↑j  
.text:00401E39 ; .text:00401DC3↑j  
.text:00401E39 mov eax, [ebp-270h]  
.text:00401E3F  
.text:00401E3F loc_401E3F: ; CODE XREF: .text:00401DD1↑j  
.text:00401E3F mov [ebp-170h], eax  
.text:00401E45  
.text:00401E45 loc_401E45: ; CODE XREF: .text:00401E2B↑j  
.text:00401E45 push dword ptr [ebp-16Ch]  
.text:00401E48 call dword ptr [ebp-34h]  
.text:00401E4E cmp dword ptr [ebp-170h], 'awmv' ;  
.text:00401E4E ; search known sandboxes'  
.text:00401E4E ; substring in registry key value  
.text:00401E4E ; vbox  
.text:00401E4E ; qemu  
.text:00401E4E ; vmwa  
.text:00401E58 jz short loc_401E95  
.text:00401E5A cmp dword ptr [ebp-170h], 'xobv'  
.text:00401E64 jz short loc_401E95  
.text:00401E66 cmp dword ptr [ebp-170h], 'umeq'  
.text:00401E70 jz short loc_401E95  
.text:00401E72  
.text:00401E72 loc_401E72: ; CODE XREF: .text:00401D55↑j  
.text:00401E72 ; .text:00401D6D↑j ...  
.text:00401F72 rdtsc
```

Detect Analysis Engine

The screenshot shows the Enigma Group's Hacking Forum interface. At the top, there are navigation links: HOME, FORUMS, EXTRA, DONATIONS, LOGIN, and REGISTER. The main content area is titled "Enigma Group's Hacking Forum" and features a "User Info" section with a welcome message for a guest user, a login form with fields for username, password, and session length, and a search bar. To the right, there are sections for "News" (promoting a hash cracker) and "Forum Stats" (showing 39005 posts, 4766 topics, and 23414 members). The main post is titled "[C++] Anti-Sandbox" and is by user "blink_212". The post text states: "This is basidly a combination of my old work, and some other code have ported over from VB. I'll release the current source for what im working on somewhere else... 😊". Below the text is a code block for a C++ function named "detectSandbox".

Enigma Group's Hacking Forum

HOME FORUMS EXTRA DONATIONS LOGIN REGISTER

User Info
Welcome, **Guest**. Please [login](#) or [register](#).
Did you miss your [activation email](#)?
January 31, 2013, 02:42:53 PM

News
Need a hash cracked? Use the Enigma Group [Hash Cracker!](#) It's the largest hash library on the interwebz.

Forum Stats
39005 Posts in 4766 Topics by
23414 Members
Latest Member: [young12dre](#)

Search: Search [Advanced search](#)

Enigma Group's Hacking Forum | [Hacking](#) | [Undetection Techniques](#) | [\[C++\] Anti-Sandbox](#)

Pages: [1] [PRINT](#)

Author: [blink_212](#) (Global Moderator, Veteran, Offline, 1438 Posts, +6 Respect, EG Fanatic.)

Topic: [\[C++\] Anti-Sandbox](#) (Read 2487 times)

[\[C++\] Anti-Sandbox](#)
on: January 28, 2011, 01:46:21 AM

This is basidly a combination of my old work, and some other code have ported over from VB. I'll release the current source for what im working on somewhere else... 😊

```
Code: [Select]
bool detectSandbox(char* exeName, char* user){
    // Used for detecting sandboxes. So far it detects
    // Arubis, CG, Sunbelt, Sandboxie, Norman, WinFail.

    char* str = exeName;
    char * pch;

    HWND snd;

    if( (snd = FindWindow("SandboxieControlWndClass", NULL)) ){
        return true; // Detected Sandboxie.
    }
}
```

Detect Analysis Engine

Enigma Group's Hacking Forum

[HOME](#) [FORUMS](#) [EXTRA](#) [DONATIONS](#) [LOGIN](#) [REGISTER](#)

```
if( (snd = FindWindow("SandboxieControlWndClass", NULL)) ){
    return true; // Detected Sandboxie.
} else if( (pch = strstr (str,"sample")) || (user == "andy") || (user == "Andy") ){
    return true; // Detected Anubis sandbox.
} else if( (exeName == "C:\file.exe") ){
    return true; // Detected Sunbelt sandbox.
} else if( (user == "currentuser") || (user == "Currentuser") ){
    return true; // Detected Norman Sandbox.
} else if( (user == "Schmidt") || (user == "schmidt") ){
    return true; // Detected CW Sandbox.
} else if( (snd = FindWindow("Afx:400000:0", NULL)) ){
    return true; // Detected WinJail Sandbox.
} else {
    return false;
}
```

Avoid Monitoring

- ▶ Open window and wait for user to click
- ▶ Only do bad things after system reboots
 - ▶ system could catch the fact that malware tried to make itself persistent
- ▶ Only run before / after specific dates

Avoid Monitoring

```
SYSTEMTIME SystemTime;

DisableThreadLibraryCalls(hdll);
GetSystemTime(&SystemTime);
result = SystemTime.wMonth;
if (SystemTime.wDay + 100 * (SystemTime.wMonth + 100 * (unsigned int)SystemTime.wYear)
    >= 20120101)
{
    uint8_t* pmain_image = (uint8_t*)GetModuleHandleA(0);
    IMAGE_DOS_HEADER *pdos_header = (IMAGE_DOS_HEADER*)pmain_image;
    IMAGE_NT_HEADERS *pnt_header = \
        (IMAGE_NT_HEADERS*) (pdos_header->e_lfanew + pmain_image);
    uint8_t* entryPoint = pmain_image + pnt_header->OptionalHeader.AddressOfEntryPoint;
    result = VirtualProtect(entryPoint, 0x10u, 0x40u, &flOldProtect);

    if (result)
    {
        entryPoint[0] = 0xE9;
        entryPoint[1] = (uint8_t) ((uint8_t *)loadShellCode - entryPoint - 5);
        entryPoint[2] = (uint8_t) (((uint8_t *)loadShellCode - entryPoint - 5) >> 8);
        entryPoint[3] = (uint8_t) (((uint8_t *)loadShellCode - entryPoint - 5) >> 16);
        entryPoint[4] = (uint8_t) (((uint8_t *)loadShellCode - entryPoint - 5) >> 24);
        result = VirtualProtect((LPVOID)entryPoint, 0x10u, flOldProtect, &flOldProtect);
    }
}
```


Avoid Monitoring

- ▶ Escape 32-bit address space (on 64-bit Windows)
 - ▶ 32-bit Windows processes actually live in 64-bit address space
 - ▶ code can modify segment register to point outside “normal” 32-bit address space
 - ▶ Windows uses this trick to call 64-bit system calls from 32-bit code (basically, 32-bit system calls are trampolines to 64-bit versions)
 - ▶ malware uses this to bypass systems that monitor 32-bit addresses of system calls

Avoid Monitoring

- ▶ Sleep for a while (analysis systems have time-outs)
 - ▶ typically, a few minutes will do this
- ▶ “Sleep” in a smarter way (stalling code – example on the next slide)

Avoid Monitoring

```
1 unsigned count, tick;
2
3 void helper() {
4     tick = GetTickCount();
5     tick++;
6     tick++;
7     tick = GetTickCount();
8 }
9
10 void delay() {
11     count=0x1;
12     do {
13         helper();
14         count++;
15     } while (count!=0xe4e1c1);
16 }
```

Figure 1. Stalling code found in real-world malware (W32.DelfInj)

Handling Evasions



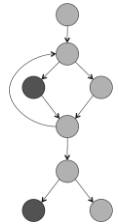
Security in knowledge



RSAC CONFERENCE
EUROPE 2013

What can we do about evasion?

- ▶ One key evasive technique relies on checking for specific values in the environment (triggers)
 - ▶ we can randomize these values, if we know about them
 - ▶ we can detect (and bypass) triggers automatically
- ▶ Another key technique relies on timing out the sandbox
 - ▶ we can automatically profile code execution and recognize stalling



Bypassing Triggers



▶ Idea

- ▶ explore multiple execution paths of executable under test
- ▶ exploration is driven by monitoring how program uses inputs
- ▶ system should also provide information under which circumstances a certain action is triggered

▶ Approach

- ▶ track “interesting” input when it is read by the program
- ▶ whenever a control flow decision is encountered that uses such input, two possible paths can be followed
- ▶ save snapshot of current process and continue along first branch
- ▶ later, revert back to stored snapshot and explore alternative branch

Bypassing Triggers



- ▶ Tracking input
 - ▶ we already know how to do this (tainting)
- ▶ Snapshots
 - ▶ we know how to find control flow decision points (branches)
 - ▶ snapshots are generated by saving the content of the process' virtual address space (of course, only used parts)
 - ▶ restoring works by overwriting current address space with stored image
- ▶ Explore alternative branch
 - ▶ restore process memory image
 - ▶ set the tainted operand (register or memory location) to a value that reverts branch condition
 - ▶ let the process continue to run

Bypassing Triggers



- ▶ Unfortunately, it is not that easy
 - ▶ when only rewriting the operand of the branch, process state can become inconsistent
 - ▶ input value might have been copied or used in previous calculations

```
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```


Bypassing Triggers



- ▶ Unfortunately, it is not that easy
 - ▶ when only rewriting the operand of the branch, process state can become inconsistent
 - ▶ input value might have been copied or used in previous calculations

```
x = 0
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```

Bypassing Triggers



- ▶ Unfortunately, it is not that easy
 - ▶ when only rewriting the operand of the branch, process state can become inconsistent
 - ▶ input value might have been copied or used in previous calculations

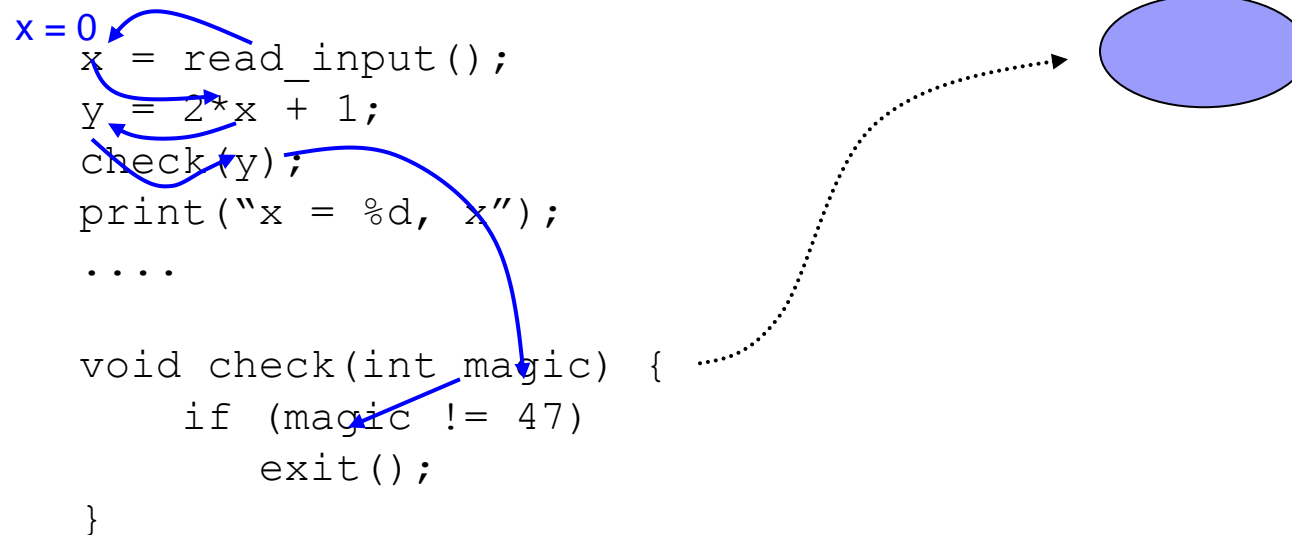
```
x = 0
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```

Bypassing Triggers



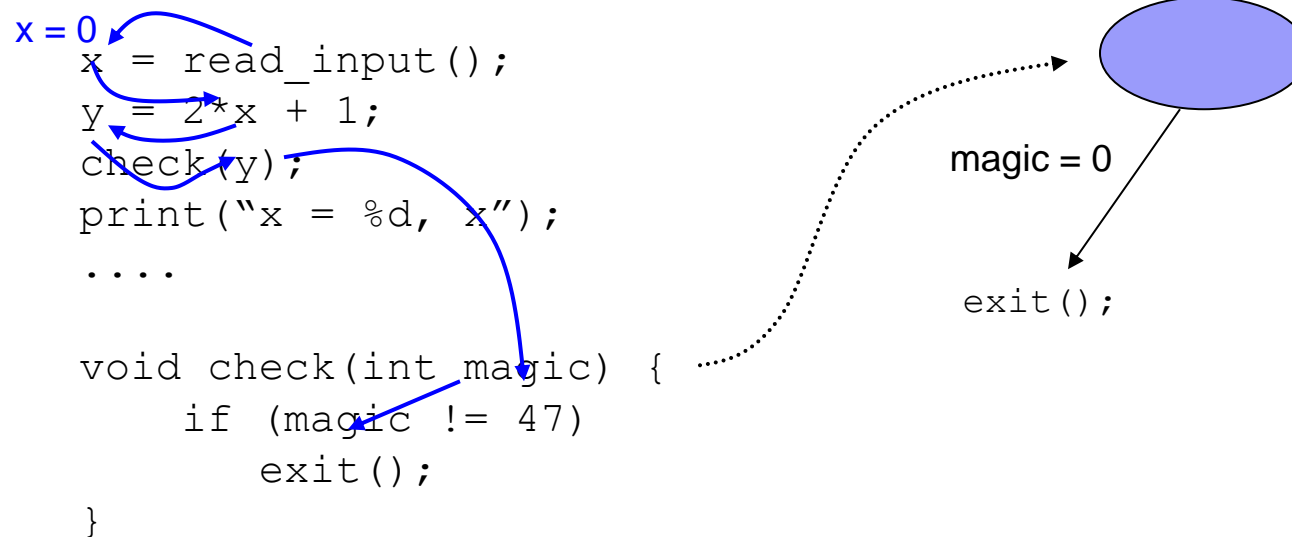
- ▶ Unfortunately, it is not that easy
 - ▶ when only rewriting the operand of the branch, process state can become inconsistent
 - ▶ input value might have been copied or used in previous calculations



Bypassing Triggers



- ▶ Unfortunately, it is not that easy
 - ▶ when only rewriting the operand of the branch, process state can become inconsistent
 - ▶ input value might have been copied or used in previous calculations



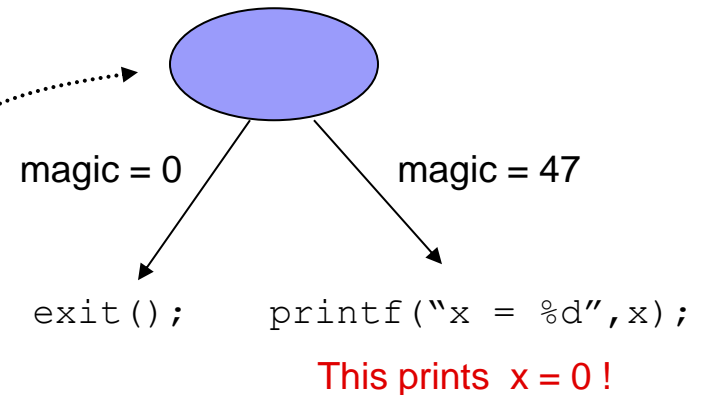
Bypassing Triggers



- ▶ Unfortunately, it is not that easy
 - ▶ when only rewriting the operand of the branch, process state can become inconsistent
 - ▶ input value might have been copied or used in previous calculations

```
x = 0
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```



We have to remember that y depends on x , and that $magic$ depends on y .

Bypassing Triggers



- ▶ Tracking of input must be extended
 - ▶ whenever a tainted value is copied to a new location, we must remember this relationship
 - ▶ whenever a tainted value is used as input in a calculation, we must remember the relationship between the input and the result
- ▶ Constraint set
 - ▶ for every operation on tainted data, a constraint is added that captures relationship between input operands and result
 - ▶ can be used to perform consistent memory updates when exploring alternative paths
 - ▶ provides immediate information about condition under which path is selected

Bypassing Triggers



▶ Constraint Set

```
x = read_input();  
y = 2*x + 1;  
check(y);  
print("x = %d, x");  
.....
```

```
void check(int magic) {  
    if (magic != 47)  
        exit();  
}
```

Bypassing Triggers



► Constraint Set

```
x = 0
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```

```
x == input
y == 2*x + 1
magic == y
```


Bypassing Triggers



► Constraint Set

```
x = 0  
x = read_input();  
y = 2*x + 1;  
check(y);  
print("x = %d, x");  
.....
```

```
void check(int magic) {  
    if (magic != 47)  
        exit();  
}
```

```
x == input  
y == 2*x + 1  
magic == y  
magic == 47
```

Bypassing Triggers



► Constraint Set

```
x = 0  
x = read_input();  
y = 2*x + 1;  
check(y);  
print("x = %d, x");  
.....
```

```
void check(int magic) {  
    if (magic != 47)  
        exit();  
}
```

```
x == input  
y == 2*x + 1  
magic == y  
magic == 47
```

solve for alternative
branch

```
y == magic == 47  
x == input == 23
```

Now, print outputs "x = 23"

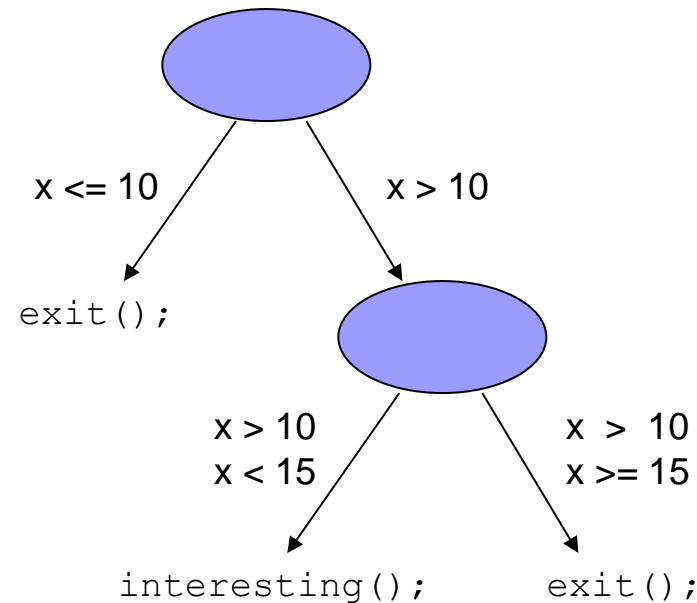
Bypassing Triggers



▶ Path constraints

- ▶ capture effects of conditional branch operations on tainted variables
- ▶ added to constraint set for certain path

```
x = read_input();  
  
if (x > 10)  
    if (x < 15)  
        interesting();  
  
exit();
```



Bypassing Triggers



- ▶ 308 malicious executables
 - ▶ large variety of viruses, worms, bots, Trojan horses, ...

Additional code is likely for error handling

Interesting input sources	
Check for Internet connectivity	20
Check for mutex object	116
Check for existence of file	79
Check for registry entry	74
Read current time	134
Read from file	106
Read from network	134

Additional code coverage	
none	136
0% - 10%	21
10% - 50%	71
50% - 200%	37
> 200%	43

Relevant behavior:
time-triggers
filename checks
bot commands

Combating Evasion



▶ Mitigate stalling loops

1. detect that program does not make progress
2. passive mode
 - ▶ find loop that is currently executing
 - ▶ reduce logging for this loop (until exit)
3. active mode
 - ▶ when reduced logging is not sufficient
 - ▶ actively interrupt loop

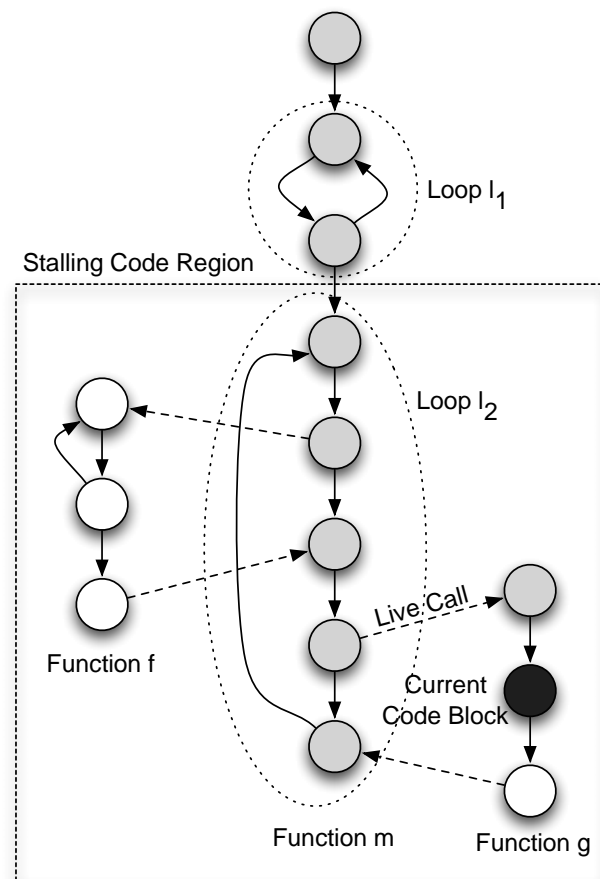
▶ Progress checks

- ▶ based on system calls
- ▶ too many failures, too few, always the same, ...

Passive Mode



- ▶ Finding code blocks (white list) for which logging should be reduced
 - ▶ build dynamic control flow graph
 - ▶ run loop detection algorithm
 - ▶ identify live blocks and call edges
 - ▶ identify first (closest) active loop (loop still in progress)
 - ▶ mark all regions reachable from this loop



Active Mode



- ▶ Interrupt loop
 - ▶ find conditional jump that leads out of white-listed region
 - ▶ simply invert it the next time control flow passes by

- ▶ Problem
 - ▶ program might later use variables that were written by loop but that do not have the proper value and fail

- ▶ Solution
 - ▶ mark all memory locations (variables) written by loop body
 - ▶ dynamically track all variables that are marked (taint analysis)
 - ▶ whenever program uses such variable, extract slice that computes this value, run it, and plug in proper value into original execution

Experimental Results



Description	# samples	%	# AV families
<i>base run</i>	29,102	—	1329
<i>stalling</i>	9,826	33.8%	620
<i>loop found</i>	6,237	21.4%	425

- 1,552 / 6,237 stalling samples reveal additional behavior
- At least 543 had obvious signs of malicious (deliberate) stalling

Description	Passive			Active		
	# samples	%	# AV families	# samples	%	# AV families
<i>Runs total</i>	3,770	—	319	2,467	—	231
<i>Added behavior (any activity)</i>	1,003	26.6%	119	549	22.3%	105
- Added file activity	949	25.2%	113	359	14.6%	79
- Added network activity	444	11.8%	52	108	4.4%	31
- Added GUI activity	24	0.6%	15	260	10.5%	51
- Added process activity	499	13.2%	55	90	3.6%	41
- Added registry activity	561	14.9%	82	184	7.5%	52
- Exception cases	21	0.6%	13	273	11.1%	48
<i>Ignored (possibly random) activity</i>	1,447	38.4%	128	276	11.2%	72
- Exception cases	0	0.0%	0	82	3.3%	27
<i>No new behavior</i>	1,320	35.0%	225	1,642	66.6%	174
- Exception cases	0	0.0%	0	277	11.2%	63

Conclusions

- ▶ Malware is key component in many security threats on the Internet
- ▶ Automated analysis of malicious code faces number of challenges
 - ▶ evasion is one critical challenge!
- ▶ Types of evasion
 - ▶ detect analysis environment
 - ▶ detect analysis system
 - ▶ avoid analysis
- ▶ We shouldn't simply give up, it is possible to address certain techniques in very general ways



Security in knowledge

Thank you!

Christopher Kruegel

Lastline Inc. / UCSB

chris@lastline.com

<http://www.lastline.com>