BlackBox

Lightweight Security Monitoring for COTS Binaries

Byron Hawkins and Brian Demsky
University of California, Irvine, USA

Michael B. Taylor
University of California, San Diego, USA
Why Security Monitoring?

Motivation #1: Exploits occur frequently.

• Thousands of vulnerabilities reported in 2015.
• Millions of new malware released every year.
• Over 90% of attacks target 10+ year old bugs.
• Windows XP still claims over 10% market share!
Why Security Monitoring?

Motivation #2: Exploit recovery requires information.

- Identify which machines were affected:
  - Repair corrupted files.
  - Restore failed or crippled services.
  - Remove persistent malware.
- Learn how the attacker gained control.
- Prevent recurrence of the same exploit.
Why Security Monitoring?

Motivation #3: Automated security may never be feasible.
Why Security Monitoring?

Motivation #3: Automated security may never be feasible.

• COTS rewriting approaches have been defeated.
Why Security Monitoring?

Motivation #3: Automated security may never be feasible.

• COTS rewriting approaches have been defeated.
• Recompilation required, but opportunity is limited:
  • Slow adoption of security tools
  • Legacy platforms maintain significant market share
Why Security Monitoring?

Motivation #3: Automated security may never be feasible.

• COTS rewriting approaches have been defeated.
• Recompilation required, but opportunity is limited:
  • Slow adoption of security tools
  • Legacy platforms maintain significant market share
• Is it possible to build security into the compiler?
Exploit Example: Buffer Overflow

Parse the argument by copying up to the space into the buffer.

```c
static const char *
parse_option(const char *input, option_t **option_out)
{
    char buffer[8];

    for (int i = 0; input[i] != '\0'; i++) {
        if (input[i] != ' ') {
            buffer[i] = input[i];
        } else {
            ...  
        }
    }
}
```

$ foobar limit 3
limit 3
2.83749
Exploit Example: Buffer Overflow

User input can change the return address of this function!

```
$ foobar option-with-inval
Segmentation fault

buffer: <base pointer>

static const char *
parse_option(const char *input, option_t **option_out)
{
    char buffer[8];

    for (int i = 0; ...) {
        ...
    }
}
```

0x00007ffffffffd418  arg #2: **option_out
0x00007ffffffffdae7  arg #1: *input
0x0000000100000000  local #2: i
0x772d6e6f6974706f  local #1: buffer
0x61766e692d687469 <previous stack frame base>
0x756c61762d64696c <return address>

Adversary controls the program!
Automated Security

Defense concept:

- Identify vulnerable program instructions

```
0x00007fffffffd418  arg #2:  **option_out
0x00007ffffffffdae7  arg #1:  *input
0x0000000100000000  local #2:  i
0x772d6e6f6974706f  local #1:  buffer
0x61766e692d687469    <previous stack frame base>
0x756c61762d64696c    <return address>
```

Adversary controls the program!
Automated Security

Defense concept:

- Identify vulnerable program instructions
- Detect adversarial manipulation of instruction operands

Adversary controls the program!
Automated Security

Defense problem:

• How to distinguish adversarial influence?

Adversary controls the program!
Automated Security

Defense proposal:

• Detect unintended data flows...

Adversary controls the program!
Automated Security

Defense proposal:

- Detect unintended data flows...
  - Intended data flows can be exploited!

Adversary controls the program!
Automated Security

Defense proposal:

- Detect unintended data flows...
- Intended data flows can be exploited!
- Calculate intended operand values
  - e.g. legitimate return addresses

```
0x00007fffffffffd418  arg #2: **option_out
0x00007ffffffffdae7   arg #1:    *input
0x0000000010000000     local #2:  i
0x00007fffffff7a25161  local #1:    buffer
0x00007fffffffffd420   <previous stack frame base>
0x00000000400761       <return address>
```

main()
Automated Security

Defense proposal:

- Detect unintended data flows...
  - Intended data flows can be exploited!
- Calculate intended operand values
  - e.g. legitimate return addresses
- Counter example: Microsoft media licensing

```c
0x00007fffffff418  arg #2:  **option_out
0x00007fffffffdae7  arg #1:  *input
0x0000000100000000  local #2:  i
0x00007fffffff7a25161  local #1:  buffer
0x00007fffffff420  <previous stack frame base>
0x0000000000400761  <return address>

main()
```
Automated Security Limitations

At startup of Microsoft Word, a media license module dynamically generates a small routine on the heap:

- **ipcsecproc.dll**: Code generator
- **DGC**: 56 basic blocks
Automated Security Limitations

It pushes a placeholder on the call stack...

ipcsecproc.dll

create fake stack frame

Call Stack

heap data

DGC
56 basic blocks
Automated Security Limitations

...followed by a non-conventional “return” to enter the DGC.
Automated Security Limitations

Calculating this return address requires calculating heap states.

ipcsecproc.dll

Call Stack

DGC
56 basic blocks

"incorrect" return

heap data
Automated Security Limitations

The DGC routine calls several security-sensitive functions.

- VirtualProtect()
- DeviceIoControl()
- CreateFile()
- "incorrect" return
Automated Security Limitations

Defense proposal:

- Detect unintended data flows...
  - Intended data flows can be exploited!
- Calculate intended operand values
  - e.g. legitimate return addresses
- Counter example: Microsoft media licensing
- Counter example: Windows thread injection
Automated Security Limitations

DWORD yourAppProcessId = GetProcessByName("YourApp.exe");
Automated Security Limitations

DWORD yourAppProcessId = GetProcessByName("YourApp.exe");

HANDLE yourAppProcess = OpenProcess(PROCESS_ALL_ACCESS, yourAppProcessId);
Automated Security Limitations

DWORD yourAppProcessId = GetProcessByName("YourApp.exe");
HANDLE yourAppProcess = OpenProcess(PROCESS_ALL_ACCESS, yourAppProcessId);
HMODULE kernel32 = GetModuleHandle("kernel32.dll");
Automated Security Limitations

DWORD yourAppProcessId = GetProcessByName("YourApp.exe");
HANDLE yourAppProcess = OpenProcess(PROCESS_ALL_ACCESS, yourAppProcessId);
HMODULE kernel32 = GetModuleHandle("kernel32.dll");
LPTHREAD_START_ROUTINE loadLibrary = GetProcAddress(kernel32, "LoadLibraryA");
Automated Security Limitations

DWORD yourAppProcessId = GetProcessByName("YourApp.exe");
HANDLE yourAppProcess = OpenProcess(PROCESS_ALL_ACCESS, yourAppProcessId);
HMODULE kernel32 = GetModuleHandle("kernel32.dll");
LPTHREAD_START_ROUTINE loadLibrary = GetProcAddress(kernel32, "LoadLibraryA");
HANDLE hThread = CreateRemoteThread(yourAppProcess, loadLibrary, myMalwareDllPath);
Why Security Monitoring?

Motivation #3: Automated security may never be feasible.

• COTS rewriting approaches have been defeated.
• Recompilation required, but opportunity is limited:
  • Slow adoption of security tools
  • Legacy platforms maintain significant market share
• Indirect branch targets may be difficult to calculate:
  • Microsoft media licensing
  • Windows thread injection
Goals of BlackBox

• Log malicious and abnormal program activity
• Minimize logging of ordinary control flow
• Block known exploits and protect known vulnerabilities
Naïve Application Monitoring

Log all branches executed by the monitored program.
Naïve Application Monitoring

**Problem:** Adversary can hide evidence after a successful exploit.
Component: **Remote Logging**

Write logs over a TCP socket to a remote server.
Component: Remote Logging

**Problem:** Billions of branch history events per hour.
Component: **Binary Translation**

Execute code from a cached copy, logging each CFG edge once.
Next call to `foo()` will not log A → C → D → E again.
Component: Binary Translation

Problem: Still up to millions of CFG edges per hour.
Component: Trusted Profile

Learn normal program behavior during offline profiling.
Offline Profiling

Record

Monitor

Trustee Trace

Merge

Trusted Profile

**BlackBox**

A: cmp ax bx
   jge ax
B: add ax cx
   jmp ax
C: add bx cx
D: sub dx cx
Component: Trusted Profile

Filter trusted CFG edges before logging.
Component: Trusted Profile

Shadow Stack: Always trust conventional return edges.
Tusting Dynamic Code

1. Large desktop applications typically use JIT engines:

<table>
<thead>
<tr>
<th>Application</th>
<th>JIT Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Office</td>
<td>JScript9</td>
</tr>
<tr>
<td>Adobe PDF Reader</td>
<td>JScript9, ShockwaveFlash</td>
</tr>
<tr>
<td>Chrome</td>
<td>V8, PepperFlash</td>
</tr>
</tbody>
</table>
T锃g Dynamic Code

1. Large desktop applications typically use JIT engines:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Office</td>
<td>JScript9</td>
</tr>
<tr>
<td>Adobe PDF Reader</td>
<td>JScript9, ShockwaveFlash</td>
</tr>
<tr>
<td>Chrome</td>
<td>V8, PepperFlash</td>
</tr>
</tbody>
</table>

2. Observationally equivalent executions frequently exhibit low-level differences in generated code.
Component: Dynamic Code Abstraction

JIT engines may generate millions of basic blocks.
Component: Dynamic Code Abstraction

Random factors affect low-level code, e.g.:
- OS thread scheduling
- Webserver response time
- Internal timers
Component: Dynamic Code Abstraction

**JIT Profile:** Learn trusted entry and exit points (API).
Component: Dynamic Code Abstraction

Encapsulate JIT code in a singleton CFG node.
Component: Dynamic Code Abstraction

Learn the trusted API for the JIT code.
Improve System Call Visibility

- Exploits rely on system calls to carry out malicious actions such as creating or deleting files.
Improve System Call Visibility

• Exploits rely on system calls to carry out malicious actions such as creating or deleting files.

• System calls are usually invoked from trampolines in core system libraries like ntdll:

```
[syscall 0x52]
ntdll!ZwCreateFile()
KERNELBASE!CreateFileW()
kernel32!CreateFileWImplementation()
DWrite!File::File()
DWrite!LocalFileLoader::FontFileStream()
DWrite!LocalFileLoader::CreateFontFileStream()
```

```
[syscall 0x52]
ntdll!ZwCreateFile()
KERNELBASE!CreateFileW()
KERNELBASE!BasepLoadLibraryAsDataFileInternal()
KERNELBASE!BasepLoadLibraryAsDataFile()
KERNELBASE!LoadLibraryExW()
KERNELBASE!ConvertTimeZoneMuiString()
```
Improve System Call Visibility

- Exploits rely on system calls to carry out malicious actions such as creating or deleting files.
- System calls are usually invoked from trampolines in core system libraries like ntdll:

  ```
  [syscall 0x52]
  ntdll!ZwCreateFile()
  KERNELBASE!CreateFileW()
  kernel32!CreateFileWImplementation()
  DWrite!File::File()
  DWrite!LocalFileLoader::FontFileStream()
  DWrite!LocalFileLoader::CreateFontFileStream()
  [syscall 0x52]
  ntdll!ZwCreateFile()
  KERNELBASE!CreateFileW()
  KERNELBASE!BasepLoadLibraryAsDataFileInternal()
  KERNELBASE!BasepLoadLibraryAsDataFile()
  KERNELBASE!LoadLibraryExW()
  KERNELBASE!ConvertTimeZoneMuiString()
  ```

Execution of a trusted trampoline does not always imply execution of a trusted system call.
Improve System Call Visibility

The ZwCreateFile trampoline is trusted—what can go wrong?

Load fonts from a system table:

```c
[syscall 0x52]
ntdll!ZwCreateFile()
KERNELBASE!CreateFileW()
kernel32!CreateFileWImplementation()
DWrite!File::File()
DWrite!LocalFileLoader::FontFileStream()
DWrite!LocalFileLoader::CreateFontFileStream()
DWrite!LocalFileLoader::CreateStreamFromKey()
DWrite!FontFileReference::GetStreamInternal()
DWrite!FontFragmentPtr<unsigned char>()
DWrite!FontFileReference::ReadIntoBuffer()
DWrite!OpenTypeTableDirectory::GetTableCount()
```

Load the timezone library:

```c
[syscall 0x52]
ntdll!ZwCreateFile()
KERNELBASE!CreateFileW()
KERNELBASE!BasepLoadLibraryAsDataFileInternal()
KERNELBASE!BasepLoadLibraryAsDataFile()
KERNELBASE!LoadLibraryExW()
KERNELBASE!ConvertTimeZoneMuiString()
```

A malicious font could cause a malware library to be loaded!
Improve System Call Visibility

The ZwCreateFile trampoline is trusted—what can go wrong?

```c
[syscall 0x52]
ntdll!ZwCreateFile()
KERNELBASE!CreateFileW()
KERNELBASE!BasepLoadLibraryAsDataFileInternal()
KERNELBASE!BasepLoadLibraryAsDataFile()
KERNELBASE!LoadLibraryExW()

DWrite!FontFileReference::GetStreamInternal()
DWrite!FontFragmentPtr<unsigned char>()
DWrite!FontFileReference::ReadIntoBuffer()
DWrite!OpenTypeTableDirectory::GetTableCount()
```
Component: Stack Spy

Suspicious syscall: influenced by an untrusted branch.

[syscall 0x52]
ntdll!ZwCreateFile()
KERNELBASE!CreateFileW()
KERNELBASE!BasepLoadLibraryAsDataFileInternal()
KERNELBASE!BasepLoadLibraryAsDataFile()
KERNELBASE!LoadLibraryExW()

DWrite!FontFileReference::GetStreamInternal()
DWrite!FontFragmentPtr<unsigned char>()
DWrite!FontFileReference::ReadIntoBuffer()
DWrite!OpenTypeTableDirectory::GetTableCount()
Component: Stack Spy

Suspicious syscall: influenced by an untrusted branch.

```
suspicious
  \- syscall 0x52
  \- ntdll!ZwCreateFile()
  \- KERNELBASE!CreateFileW()
  \- KERNELBASE!BasepLoadLibraryAsDataFileInternal()
  \- KERNELBASE!BasepLoadLibraryAsDataFile()
  \- KERNELBASE!LoadLibraryExW()

untrusted!
  \- DWrite!FontFileReference::GetStreamInternal()
  \- DWrite!FontFragmentPtr<unsigned char>()
  \- DWrite!FontFileReference::ReadIntoBuffer()
  \- DWrite!OpenTypeTableDirectory::GetTableCount()
```

trusted
Component: Stack Spy

Suspicious syscall: influenced by an untrusted branch.

Log this syscall 0x52 as suspicious, along with the untrusted branch.
BlackBox Log

What’s in the log file?
BlackBox Log

Log sample of Adrenalin Media Player:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>900</td>
<td>Suspicious syscall #36 NtSetInformationFile adrenalinx.dll(0xf160a) -&gt; 0x97ac0 raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #36 NtSetInformationFile adrenalinx.dll(0xf160a) -&gt; 0x97ac0 raised suspicion</td>
</tr>
<tr>
<td>300</td>
<td>Structural indirect mp3dmod!CMpgaDecoder::DecodeFrame(0x76) -&gt; qasf!&lt;callback&gt;</td>
</tr>
<tr>
<td>300</td>
<td>Structural indirect quartz!CFilterGraph::AddFilterInternal(0x91) -&gt; addicted!&lt;callback&gt;</td>
</tr>
<tr>
<td>300</td>
<td>Structural indirect mp3dmod!CMpgaDecoder::DecodeFrame(0x76) -&gt; mp3dmod!CBitStream::Reset()</td>
</tr>
<tr>
<td>300</td>
<td>Structural indirect mp3dmod!CMP3DecoderDMO::DoFlush(0x16) -&gt; qasf!&lt;callback&gt;</td>
</tr>
<tr>
<td>295</td>
<td>Structural indirect qasf!CMediaWrapperFilter::JoinFilterGraph(0x68) -&gt; quartz!&lt;callback&gt;</td>
</tr>
<tr>
<td>295</td>
<td>Structural indirect qasf!ImediaSeeking::`vcall'{40}' -&gt; quartz!&lt;callback&gt;</td>
</tr>
<tr>
<td>295</td>
<td>Structural indirect qasf!CMediaWrapperFilter::DeliverInputSample(0x47) -&gt; quartz!&lt;callback&gt;</td>
</tr>
</tbody>
</table>

Suspicious syscalls occur where the player handles a media format not covered in the Trusted Profile.
Log sample of an ROP exploit on Adrenalin Player:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>999</td>
<td>Suspicious entry into DGC adrenalinx.dll(0x16f313) → DGC(0x3bfff8) raised suspicion</td>
</tr>
<tr>
<td>999</td>
<td>Incorrect return adrenalinx.dll(0x16f313) → DGC(0x3bfff8)</td>
</tr>
<tr>
<td>999</td>
<td>Untrusted module calc.exe-1db1446a00060001</td>
</tr>
<tr>
<td>999</td>
<td>Suspicious indirect shlwapi!QISearch → comctl32!&lt;callback&gt;</td>
</tr>
<tr>
<td>999</td>
<td>Suspicious indirect ntdll!LdrEnumerateLoadedModules(0x86) → kernel32!&lt;callback&gt;</td>
</tr>
<tr>
<td>999</td>
<td>Untrusted module gdiplus.dll-1db146c800060001</td>
</tr>
<tr>
<td>999</td>
<td>Suspicious indirect kernel32!BaseThreadInitThunk(0xc) → calc!&lt;callback&gt;</td>
</tr>
<tr>
<td>998</td>
<td>DGC standalone owned by adrenalinx.dll-300010001 (4 nodes)</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #25 ZwSetInformationProcess ntdll!memcpy(0x33 → 0x170) raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #82 ZwCreateFile ntdll!memcpy(0x5c → 0x128) raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #79 ZwResumeThread kernelbase!CreateRemoteThreadEx(0x2dd) → ntdll!NtResumeThread() raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #26 ZwCreateKey user32.dll(0x16d88) → ntdll!RtlDeactivateActivationContextUnsafeFast() raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #77 ZwProtectVirtualMemory apphelp!SeiIatPatch(0x86) → ntdll!RtlDeactivateActivationContextUnsafeFast() raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #165 ZwCreateThreadEx kernelbase!CreateRemoteThreadEx(0x102) → ntdll!NtCreateThreadEx() raised suspicion</td>
</tr>
</tbody>
</table>
# BlackBox Log

**How can this exploit be prevented?**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>999</td>
<td>Suspicious entry into DGC</td>
</tr>
<tr>
<td>999</td>
<td>Incorrect return adrenalinx.dll(0x16f313) -&gt; DGC(0x3bfff8)</td>
</tr>
<tr>
<td>999</td>
<td>Untrusted module calc.exe-1db1446a00060001</td>
</tr>
<tr>
<td>999</td>
<td>Suspicious indirect shlwapi!QISearch -&gt; comctl32!&lt;callback&gt;</td>
</tr>
<tr>
<td>999</td>
<td>Suspicious indirect ntdll!LdrEnumerateLoadedModules(0x86) -&gt; kernel32!&lt;callback&gt;</td>
</tr>
<tr>
<td>999</td>
<td>Untrusted module gdiplus.dll-1db146c800060001</td>
</tr>
<tr>
<td>999</td>
<td>Suspicious indirect kernel32!BaseThreadInitThunk(0xc) -&gt; calc!&lt;callback&gt;</td>
</tr>
<tr>
<td>998</td>
<td>DGC standalone owned by adrenalinx.dll-300010001 (4 nodes)</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #25 ZwSetInformationProcess</td>
</tr>
<tr>
<td></td>
<td>ntdll!memcpy(0x33 -&gt; 0x170) raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #82 ZwCreateFile</td>
</tr>
<tr>
<td></td>
<td>ntdll!memcpy(0x5c -&gt; 0x128) raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #79 ZwResumeThread</td>
</tr>
<tr>
<td></td>
<td>kernelbase!CreateRemoteThreadEx(0x2dd) -&gt; ntdll!NtResumeThread() raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #26 ZwCreateKey</td>
</tr>
<tr>
<td></td>
<td>user32.dll(0x16d88) -&gt; ntdll!RtlDeactivateActivationContextUnsafeFast() raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #77 ZwProtectVirtualMemory</td>
</tr>
<tr>
<td></td>
<td>apphelp!SeiTabPatch(0x86) -&gt; ntdll!RtlDeactivateActivationContextUnsafeFast() raised suspicion</td>
</tr>
<tr>
<td>900</td>
<td>Suspicious syscall #165 ZwCreateThreadEx</td>
</tr>
<tr>
<td></td>
<td>kernelbase!CreateRemoteThreadEx(0x102) -&gt; ntdll!NtCreateThreadEx() raised suspicion</td>
</tr>
</tbody>
</table>
Blacklist

Log entries can be converted into blacklist entries:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>999</td>
<td>Suspicious entry into DGC</td>
</tr>
<tr>
<td></td>
<td>adrenalinx.dll(0x16f313) -&gt; DGC(0x3bfffb) raised suspicion</td>
</tr>
<tr>
<td>999</td>
<td>Incorrect return adrenalinx.dll(0x16f313) -&gt; DGC(0x3bfffb)</td>
</tr>
<tr>
<td>999</td>
<td>Untrusted module calc.exe-1db1446a00060001</td>
</tr>
<tr>
<td>999</td>
<td>Suspicious indirect shlwapi!QISearch -&gt; comctl32!&lt;callback&gt;</td>
</tr>
<tr>
<td>999</td>
<td>Suspicious indirect ntdll!LdrEnumerateLoadedModules(0x86) -&gt; kernel32!&lt;callback&gt;</td>
</tr>
</tbody>
</table>

This blacklist prohibits the exploited basic block from ever making an entry into dynamic code:

```
edge adrenalinx.dll 0x16f313 <export> 0x3bfffb658587ad92
```
Blacklist

Prohibit the exploited basic block from entering DGC:

gedge adrenalinx.dll 0x16f313 <export> 0x3bfff658587ad92
Blacklist

Prohibit abnormal return sites in an entire module:

```javascript
node adrenalinx.dll <abnormal-return>
```
Blacklist

Prohibit abnormal return sites in the whole program:

node * <abnormal-return>
Blacklist

Prohibit abnormal return sites in the whole program:

\[ \text{node} \ast <\text{abnormal-return}> \]

Not viable for all programs!
- Adobe PDF Reader plugins contain abnormal returns
Benchmark: SPEC CPU 2006

Geometric Mean: 14.5%
**Problem:** Return address points to F on the left.
Correlate indirect branch targets via hashtable.
Hot paths are compiled into traces (10% speedup).
Overhead corresponds roughly to cumulative indirect branch degree.
# Log Reduction

<table>
<thead>
<tr>
<th>Program</th>
<th>All Branches</th>
<th>Binary Translation</th>
<th>Trusted Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome</td>
<td>485,251,278,660</td>
<td>6,137,106</td>
<td>7</td>
</tr>
<tr>
<td>Adobe PDF</td>
<td>34,075,711,128</td>
<td>2,292,342</td>
<td>4</td>
</tr>
<tr>
<td>Word</td>
<td>603,491,452,236</td>
<td>580,655</td>
<td>24</td>
</tr>
<tr>
<td>PowerPoint</td>
<td>251,845,377,624</td>
<td>1,335,817</td>
<td>50</td>
</tr>
<tr>
<td>Excel</td>
<td>198,427,776,372</td>
<td>561,401</td>
<td>28</td>
</tr>
<tr>
<td>Outlook</td>
<td>547,678,615,056</td>
<td>615,708</td>
<td>4</td>
</tr>
<tr>
<td>SciTE</td>
<td>61,325,719,872</td>
<td>124,013</td>
<td>33</td>
</tr>
<tr>
<td>pldflatex</td>
<td>23,504,352,560</td>
<td>64,290</td>
<td>43</td>
</tr>
<tr>
<td>Notepad++</td>
<td>129,695,545,404</td>
<td>589,155</td>
<td>24</td>
</tr>
<tr>
<td>Adrenalin</td>
<td>48,881,533,212</td>
<td>791,847</td>
<td>603</td>
</tr>
<tr>
<td>mp3info</td>
<td>2,080,031,200</td>
<td>4,339,200</td>
<td>3</td>
</tr>
</tbody>
</table>

Total log entries in one hour of normal program use.
Related Work: Other Defenses

• Code Pointer Integrity, *Kuznetsov et al*, USENIX '14
• SafeDispatch, *Jang et al*, NDSS '14
• Inference of Peak Density of Indirect Branches, *Tymburiba et al*, CGO '16
• Automated Software Diversity, *Larsen et al*, S&P '14
  • Librando, *Homescu et al*, CCS '13
• Mining Sandboxes, *Jamrozik et al*, ICSE '16
Related Work: CFI

• Control-Flow Integrity, Abadi et al, CCS '05
• Control Data Isolation, Arthur et al, CGO '15
• Context-Sensitive CFI, van der Veen et al, CCS '15
• Cryptographic CFI, Mashtizadeh et al, CCS '15
• Opaque CFI, Mohan et al, NDSS '15
• RockJIT, Niu et al, CCS '14
Related Work: Other Defenses

• Code Pointer Integrity, Kuznetsov et al, USENIX '14
• SafeDispatch, Jang et al, NDSS '14
• Inference of Peak Density of Indirect Branches, Tymburiba et al, CGO '16
• Automated Software Diversity, Larsen et al, S&P '14
  • Librando, Homescu et al, CCS '13
• Mining Sandboxes, Jamrozik et al, ICSE '16
Related Work: Attacks

• Control Flow Bending, Carlini et al, USENIX '15
• Control Flow Jujutsu, Evans et al, CCS '15
• Counterfeit Object Oriented Programming, Schuster et al, S&P '15
• Losing Control, Conti et al, CCS '15
Conclusion

• Software exploits are an ongoing problem.
• Automated security continues to face challenges.
• BlackBox provides an effective alternative to both automated security and anti-virus:
  • Isolates the pivotal actions of exploits.
  • Prevents repeated and foresee exploits.