Finding the Weak Link in Binaries

Ollie Whitehouse
Agenda

• What
• Why
• How
• Conclusions
What?
What?

Without debug symbols or source code identify Windows binaries that do not leverage the available defenses ... easily and quickly.
What?

- OS provided defenses
- Compiler provided defenses
- Compiler enabled defenses
- Linker enabled defenses
- Developer enabled defenses
- Developer secure coding practices
What?

• Version of compiler / linker
• Compiler / linker enabled protections
  • ASLR
  • DEP (NX)
  • Stack cookies
  • Safe Structured Exception Handling
• Developer used defensive APIs
  • Heap corruption behavior, DEP policy
  • DLL planting, pointer encoding
What?

• SDL banned APIs
• Dangerous APIs
  • undermining compiler/linker protections
• UAC / Integrity Level - Developer
• .NET security - Developer
  • Unmanaged code
  • Strong names
  • Partially trusted callers
Why?
Why? - Defensive

• A product == many vendors
  • e.g. Adobe Reader 10.0 == [guess?]  
• License != source code  
• License != private symbols  
• SDL assurance…  
  • getting the free security features enabled  
• End user assurance / threat awareness  
  • Understanding where you need EMET
Or put another way

- A vendor's SDL is not enough
  - doesn’t always flow upstream
- A vendor who ships doesn’t assure
  - all third party components
- End user organisations taking ownership
  - of risk
  - of mitigations
Why? - Offensive

• Mitigations are expensive / difficult
• Application specific bugs are expensive
• Maximize research ROI
  • if your goal is to exploit
  • … find the weak link
  • … reduce headaches
Or put another way

- IIS 7.5 FTP DoS
- Chris Valasek / Ryan Smith school us
  - ‘Modern Heap Exploitation using the Low Fragmentation Heap’
- Achieved EIP
  - … still no win … ASLR
  - … try an minimize the need for info leaks …
  - … lets minimize the tears …
  - … unless you want to info leak to win …
How?
Version of Compiler / Linker

• Linker version in the PE header

```
<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>MajorLinkerVersion</th>
<th>The linker major version number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>MinorLinkerVersion</td>
<td>The linker minor version number.</td>
</tr>
</tbody>
</table>
```

• ‘Rich’ header
  • Microsoft compiler specific
  • documented in 29a virus e-zine in 2004
  • further documented in 2008
  • embeds compiler IDs
  • XOR encoded
// Extract the XOR key
if (bFound == true)
{
    XORKey = binReader.ReadUInt32();
    intPos += sizeof(UInt32);

    // Now find the start of the version numbers
    int intCount2 = 0;
    int intPos2 = 0;
    bool bFound2 = false;
    UInt32 intTemp = 0;
    binReader.BaseStream.Seek(0, SeekOrigin.Begin);
    while (intCount2 < intCount && intPos < binReader.BaseStream.Length)
    {
        intTemp = binReader.ReadUInt32();
        intTemp ^= XORKey;
        //Console.WriteLine(intTemp.ToString());

        if (intTemp == 0x536E6144)
        {
            //Console.WriteLine("2 - " + intCount2.ToString());
            bFound2 = true;
            break;
        }
    }

    intPos2 += sizeof(UInt32);
    intCount2++;
}
Version of Compiler / Linker

• Version mapping exercise undertaken in January 2010
• Visual Studio 6 -> Visual Studio 2010 mapped
• Why?
  • Missing compiler protections
  • Weaker compiler protections
Compiler / Linker Protections

- **ASLR compatibility – PE header**
  
<table>
<thead>
<tr>
<th>IMAGE_DLL_CHARACTERISTICS_</th>
<th>0x0040</th>
<th>DLL can be relocated at load time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNAMIC_BASE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Data Execution Prevention – PE header**
  
<table>
<thead>
<tr>
<th>IMAGE_DLL_CHARACTERISTICS_</th>
<th>0x0100</th>
<th>Image is NX compatible.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_COMPAT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*always on for 64bit no matter what*
Compiler / Linker Protections

• Stack Cookies – PE Header, Imports and Heuristics

<table>
<thead>
<tr>
<th>60/88</th>
<th>4/8</th>
<th>SecurityCookie</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A pointer to a cookie that is used by Visual C++ or GS implementation.</td>
</tr>
</tbody>
</table>

• imports
  • _crt_debugger_hook

• heuristics – GS function epilogue / prologue
  • allows versioning
  • using FLIRT like signatures
## Compiler / Linker Protections

- **SafeSEH** – PE header (32bit only)
- **SEH** == Structured Exception Handling

<table>
<thead>
<tr>
<th>IMAGE_DLLCHARACTERISTICS_NO_SEH</th>
<th>0x0400</th>
<th>Does not use structured exception (SE) handling. No SE handler may be called in this image.</th>
</tr>
</thead>
<tbody>
<tr>
<td>64/96 4/8 SEHandlerTable</td>
<td></td>
<td>[x86 only] The VA of the sorted table of RVAs of each valid, unique SE handler in the image.</td>
</tr>
<tr>
<td>68/104 4/8 SEHandlerCount</td>
<td></td>
<td>[x86 only] The count of unique handlers in the table.</td>
</tr>
</tbody>
</table>
Compiler / Linker Protections

• Load Configuration Directory size
  • If size of directory entry <> 64 then MS12-001
    • NOT the size field in the LCD!
  • Microsoft Visual C msvcr71.dll == 72
  • Anything built with Microsoft Visual C++ .NET 2003 RTM
    • surprising amount of stuff
Default Process Heap

- Default process heap executable
- PE header

<table>
<thead>
<tr>
<th>72</th>
<th>4</th>
<th>ProcessHeapFlags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Process heap flags that correspond to the first argument of the HeapCreate function. These flags apply to the process heap that is created during process startup.</td>
</tr>
</tbody>
</table>
Shared Sections

• Shared sections executable & writeable
  • PE header
  • would be mapped across processes

<table>
<thead>
<tr>
<th>Section</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE_SCN_MEM_SHARED</td>
<td>0x10000000</td>
<td>The section can be shared in memory.</td>
</tr>
<tr>
<td>IMAGE_SCN_MEM_EXECUTE</td>
<td>0x20000000</td>
<td>The section can be executed as code.</td>
</tr>
<tr>
<td>IMAGE_SCN_MEM_READ</td>
<td>0x40000000</td>
<td>The section can be read.</td>
</tr>
<tr>
<td>IMAGE_SCN_MEM_WRITE</td>
<td>0x80000000</td>
<td>The section can be written to.</td>
</tr>
</tbody>
</table>
Defensive APIs

• HeapSetInformation
  • HeapEnableTerminationOnCorruption
• SetProcessDEPPolicy
  • PROCESS_DEP_ENABLE
• EncodePointer
Banned APIs

• Microsoft SDL banned APIs
  • parse the Import Address Table
  • 145 or them
  • indication of security awareness
Dangerous APIs

- **VirtualAlloc**
  - doesn’t benefit from ASLR
  - if mapping pages executable == win
  - released `VirtualAlloc_s.h` at Recx
- **LoadLibrary**
  - if DLL planting mitigations aren’t used
DLL / Executable Planting

• Use of `LoadLibrary` / `CreateProcess`
• But doesn’t use
  • `SetDLLDirectory`
  • `SetDefaultDllDirectories`
  • `AddDllDirectory`
• There is also a registry key
  • … more on this later
UAC / Integrity Level

• In the binaries manifest

```xml
<security>
  <requestedPrivileges>
    <requestedExecutionLevel level="asyInvoker" uiAccess="false"/>
  </requestedExecutionLevel>
</requestedPrivileges>
</security>
```
.NET Security

• Strong name checks
• Allow partially trusted callers
  • AllowPartiallyTrustedCallersAttribute
## .NET Security

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>SkipVerification</td>
</tr>
<tr>
<td>Assertion</td>
<td>False</td>
</tr>
<tr>
<td>UnmanagedCode</td>
<td>False</td>
</tr>
<tr>
<td>SkipVerification</td>
<td>True</td>
</tr>
<tr>
<td>Execution</td>
<td>False</td>
</tr>
<tr>
<td>ControlThread</td>
<td>False</td>
</tr>
<tr>
<td>ControlEvidence</td>
<td>False</td>
</tr>
<tr>
<td>ControlPolicy</td>
<td>False</td>
</tr>
<tr>
<td>SerializationFormatter</td>
<td>False</td>
</tr>
<tr>
<td>ControlDomainPolicy</td>
<td>False</td>
</tr>
<tr>
<td>ControlPrincipal</td>
<td>False</td>
</tr>
<tr>
<td>ControlAppDomain</td>
<td>False</td>
</tr>
<tr>
<td>RemotingConfiguration</td>
<td>False</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>False</td>
</tr>
<tr>
<td>BindingRedirects</td>
<td>False</td>
</tr>
<tr>
<td>Action</td>
<td>RequestMinimum</td>
</tr>
<tr>
<td>Unrestricted</td>
<td>False</td>
</tr>
<tr>
<td>TypeId</td>
<td>System.Security.Privileges.SecurityPermissionAttribute</td>
</tr>
</tbody>
</table>
Windows 8 Containers

• New for Windows 8
  • a new DLL characteristic
• Manifest
  • detailing capabilities

… for more information refer to
http://recxltd.blogspot.com/2012/03/windows-8-app-container-security-notes.html
Miscellaneous

• Force Integrity
  
  | IMAGE_DLL_CHARACTERISTICS_ | 0x0080 | Code Integrity checks are enforced. |
  | FORCE_INTEGRITY            |        |                                  |

• Company
  • File Version resource section

• Signer

• Signature type
Existing tools...
Existing Tools – Looking Glass

- from Errata Security
  - http://www.erratasec.com/
- .NET Based PE Scanner
- Scans the file system or running processes
- Limitations in checks (some)
  - No /SafeSEH
  - No /GS
  - No HeapSetInformation / SetProcessDEPPolicy
Existing Tools - BinScope

• from Microsoft
• Lots of checks
  • some of what I’ve discussed, but not all!
• Some Extra
  • non-GS friendly initialization / coverage
  • ATL version and vulnerable check
• Needs private symbols!
How I did it...
Demo
Beyond binaries

• Defense in depth features via the registry
• Needs installer teams buy-in
• or after market adoption
• Image Execution Options
  • MitigationOptions
  • CWDIllegalInDllSearch
  • DisableExceptionChainValidation
But...
Even with all these...

we don’t mitigate vtable overwrites...

```cpp
#include "stdafx.h"
#include <string.h>

class Example {
    private:
        TCHAR strBuffer[11];

    public:
        void setBuffer(TCHAR *strTemp){_tcsncpy (strBuffer, strTemp,);};
        virtual void printBuffer(){
            _tcsprintf(L"buffer loc: %p\n",&strBuffer);
            _tcsprintf(L"buffer val: %s\n",strBuffer);
        }
};

int _tmain(int argc, _TCHAR* argv[])
{
    Example *ex1;
    Example *ex2;
    Example *ex3;
    ex1 = new Example;
    ex2 = new Example;
    ex3 = new Example;
    ex1->setBuffer(L"c1c2c3c4c5");
    ex2->setBuffer(argv[1]);
    ex3->setBuffer(L"memonmodewo");
    ex1->printBuffer();
    ex2->printBuffer();
    ex3->printBuffer();
    return 0;
}
```
Bonus Material - ELF

- Similar(ish) tool exists for ELF
  - `readelf && a`
  - shell script (checksec.sh @ trapkit.de)

- RPATH / RUNPATH
  - contained in a section of an ELF
  - can override library locations
  - path doesn’t exist and you can create == win
Summary / Conclusions

• First pass binaries analysis doesn’t have to be rocket science
• Help with assurance / assessment
  • for vendors and / or end organisations
• Help with target identification
  • target lower hanging fruit
  • less SDL aware components
• Without the use of symbols…
There is still more to do…

Detect the use of the /sdl switch

Thanks! Questions?

UK Offices
Manchester - Head Office
Cheltenham
Edinburgh
Leatherhead
London
Thame

North American Offices
San Francisco
Chicago
Atlanta
New York
Seattle
Boston

Australian Offices
Sydney

European Offices
Amsterdam - Netherlands
Munich – Germany
Zurich - Switzerland

Ollie Whitehouse
ollie.whitehouse@nccgroup.com