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 vendors 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

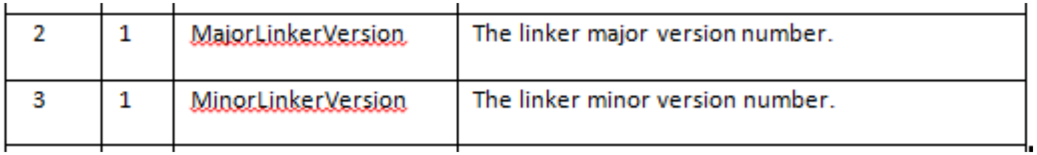
Or put another way

- 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

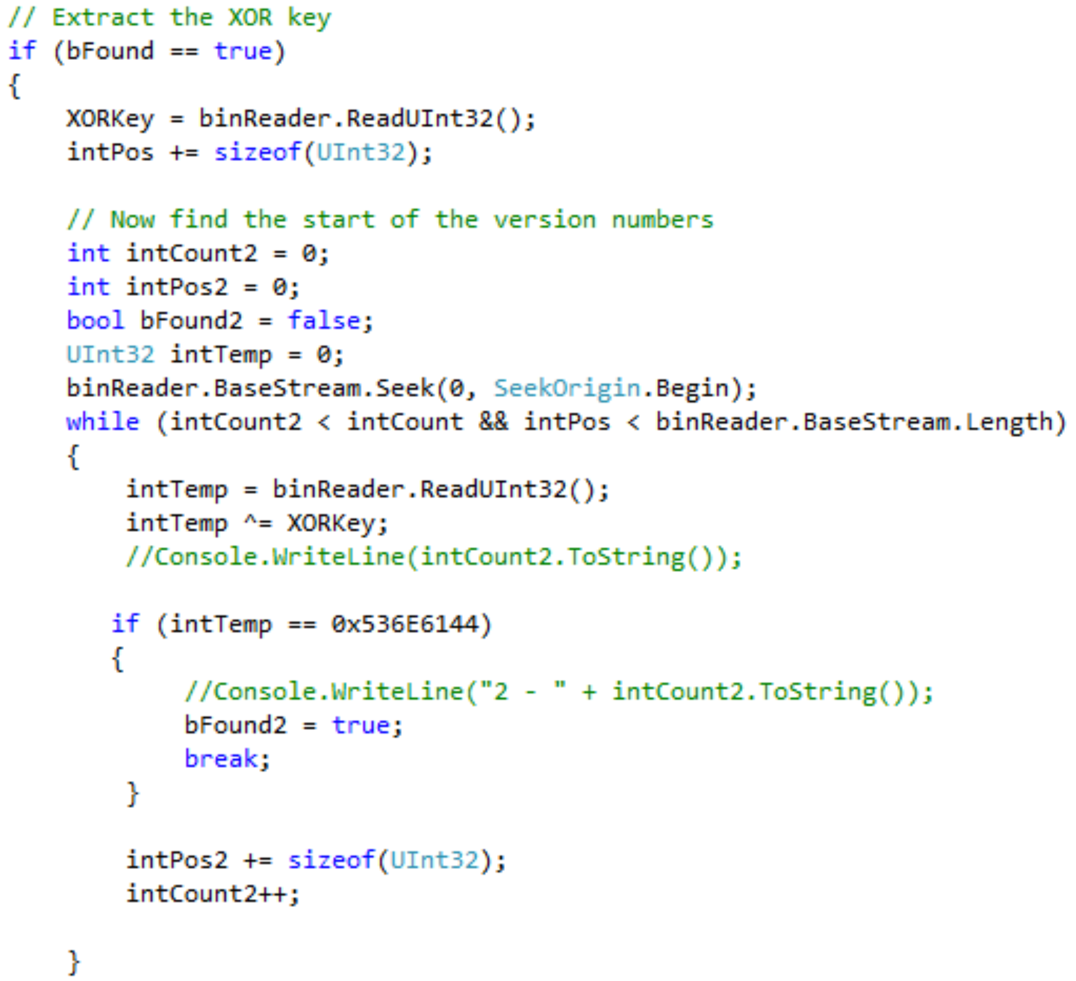
- Linker version in the PE header



2	1	MajorLinkerVersion	The linker major version number.
3	1	MinorLinkerVersion	The linker minor version number.

- ‘Rich’ header
  - Microsoft compiler specific
  - documented in 29a virus e-zine in 2004
  - further documented in 2008
  - embeds compiler IDs
  - XOR encoded

Version of Compiler / Linker



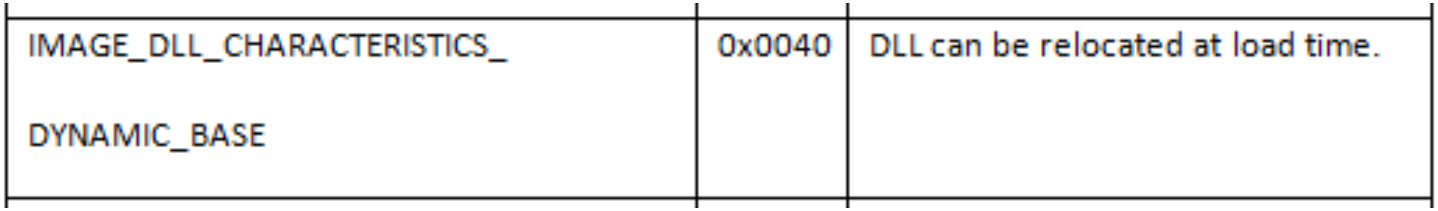
```
// Extract the XOR key
if (bFound == true)
{
    XORKey = binReader.ReadUInt32();
    intPos += sizeof(UInt32);

    // 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(intCount2.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

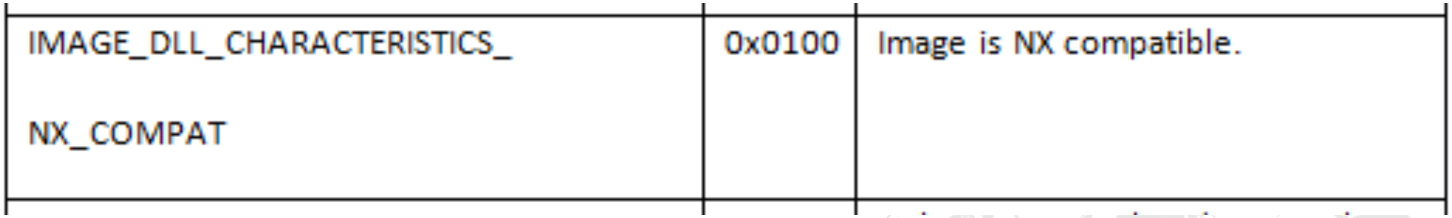
Compiler / Linker Protections

- ASLR compatibility – PE header



IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE	0x0040	DLL can be relocated at load time.

- Data Execution Prevention – PE header

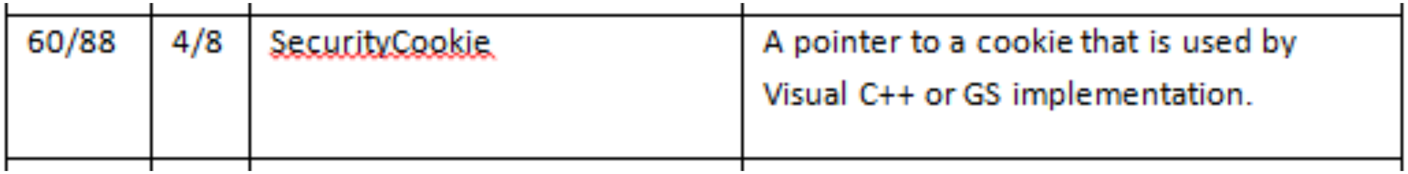


IMAGE_DLL_CHARACTERISTICS_NX_COMPAT	0x0100	Image is NX compatible.

** always on for 64bit no matter what*

Compiler / Linker Protections

- Stack Cookies – PE Header, Imports and Heuristics

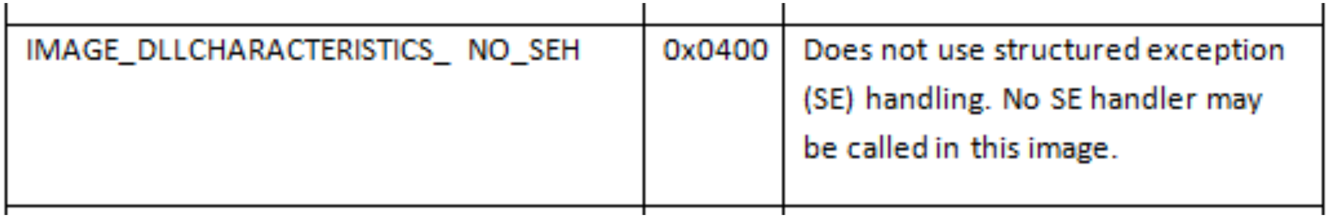


60/88	4/8	SecurityCookie	A pointer to a cookie that is used by Visual C++ or GS implementation.

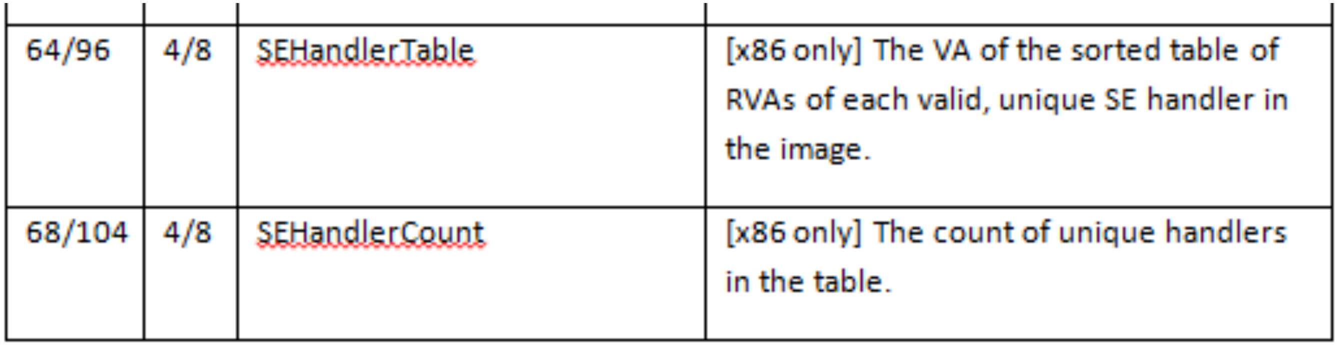
- imports
  - `_crt_debugger_hook`
- heuristics – GS function epilogue / prologue
  - allows versioning
  - using FLIRT like signatures

Compiler / Linker Protections

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



IMAGE_DLLCHARACTERISTICS_ NO_SEH	0x0400	Does not use structured exception (SE) handling. No SE handler may be called in this image.



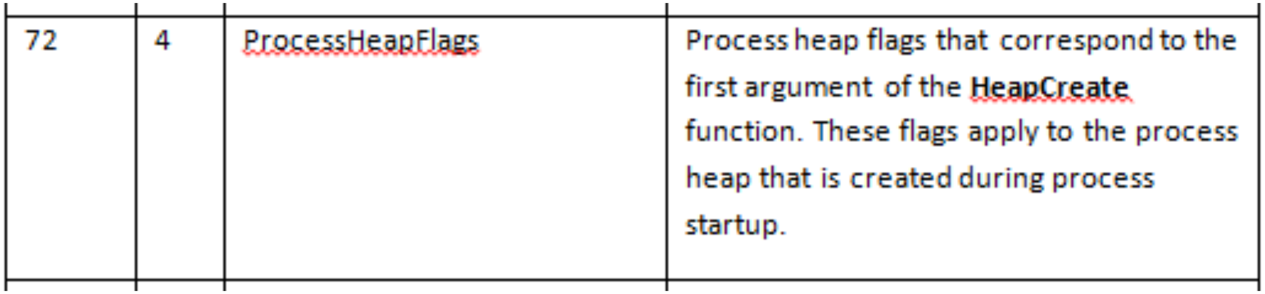
64/96	4/8	SEHandlerTable	[x86 only] The VA of the sorted table of RVAs of each valid, unique SE handler in the image.
68/104	4/8	SEHandlerCount	[x86 only] The count of unique handlers in the table.

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
    - suprising amount of stuff

Default Process Heap

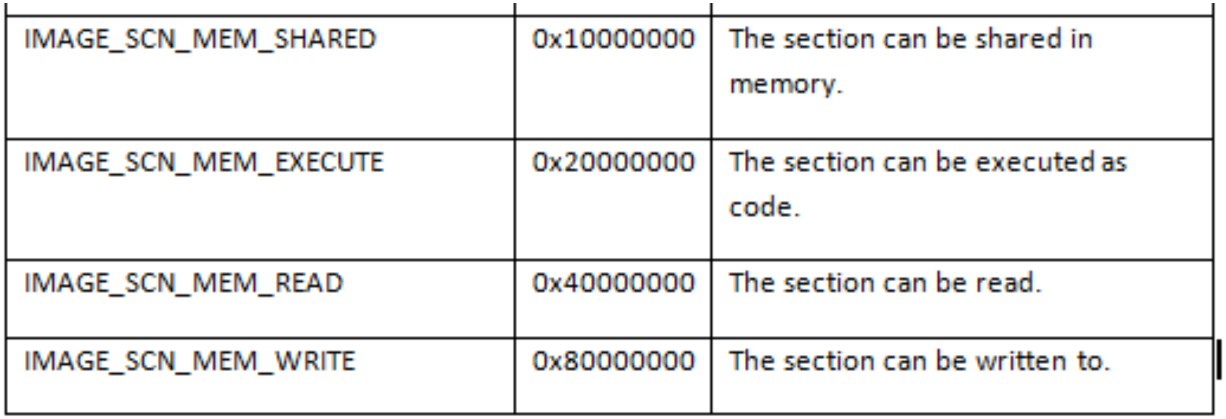
- Default process heap executable
  - PE header



72	4	ProcessHeapFlags	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.

Shared Sections

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



IMAGE_SCN_MEM_SHARED	0x10000000	The section can be shared in memory.
IMAGE_SCN_MEM_EXECUTE	0x20000000	The section can be executed as code.
IMAGE_SCN_MEM_READ	0x40000000	The section can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	The section can be written to.

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

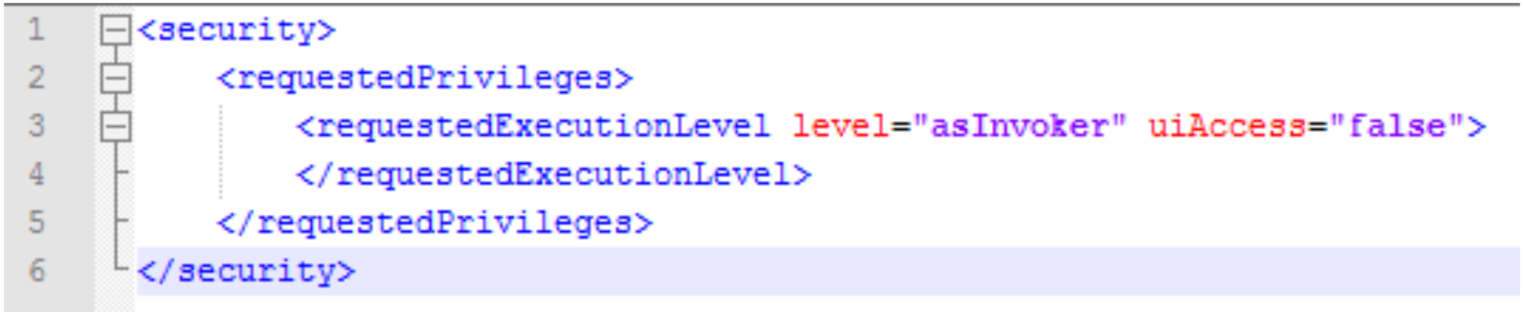
- `VirtualAlloc`
  - doesn’t benefit from ASLR
  - if mapping pages executable == win
  - released `VirtualAlloc_s.h` at Recx
- `LoadLibrary`
  - if DLL planting mitigations aren’t used

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

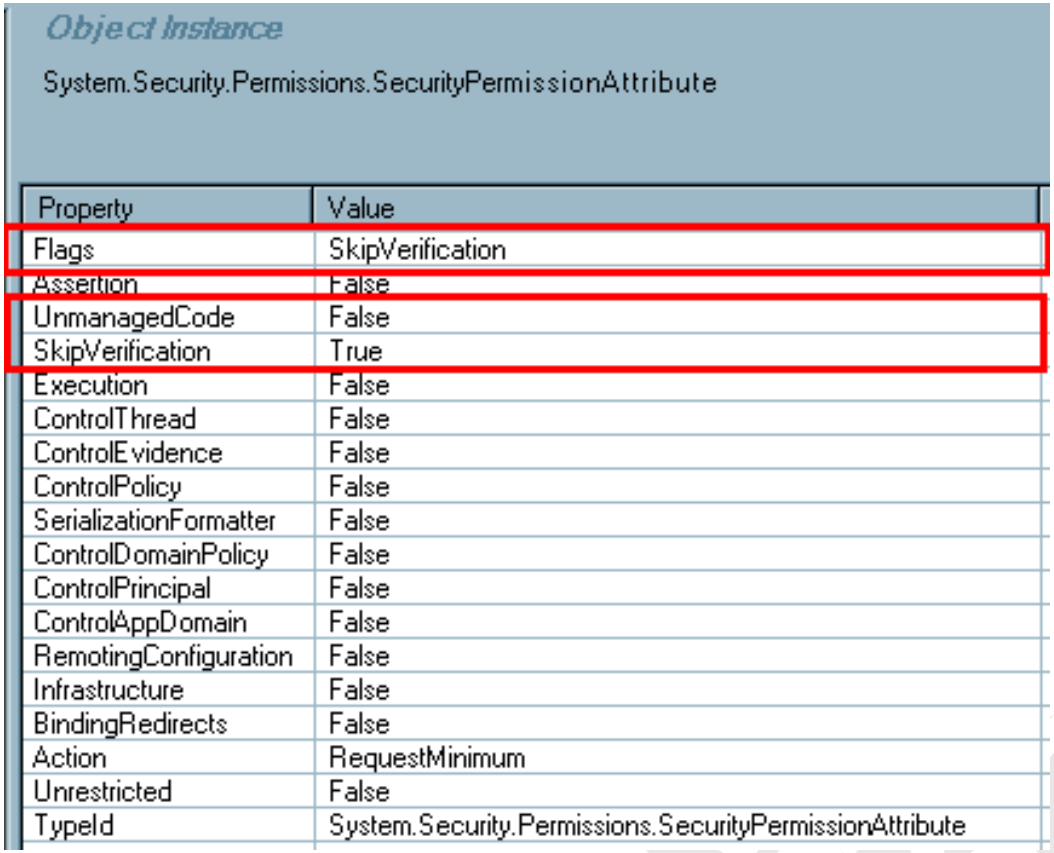


```
<security>
    <requestedPrivileges>
        <requestedExecutionLevel level="asInvoker" uiAccess="false">
        </requestedExecutionLevel>
    </requestedPrivileges>
</security>
```

.NET Security

- Strong name checks
- Allow partially trusted callers
  - `AllowPartiallyTrustedCallersAttribute`

.NET Security



Object Instance

System.Security.Permissions.SecurityPermissionAttribute

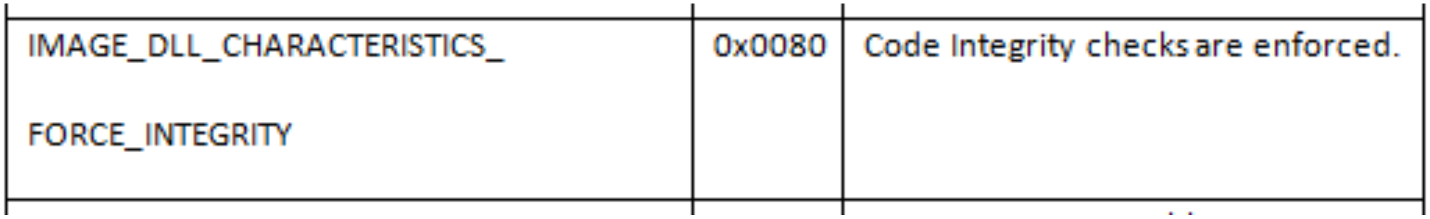
Property	Value
Flags	SkipVerification
Assertion	False
UnmanagedCode	False
SkipVerification	True
Execution	False
ControlThread	False
ControlEvidence	False
ControlPolicy	False
SerializationFormatter	False
ControlDomainPolicy	False
ControlPrincipal	False
ControlAppDomain	False
RemotingConfiguration	False
Infrastructure	False
BindingRedirects	False
Action	RequestMinimum
Unrestricted	False
TypeId	System.Security.Permissions.SecurityPermissionAttribute

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_FORCE_INTEGRITY	0x0080	Code Integrity checks are enforced.

- 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
  - http://www.microsoft.com/download/en/details.aspx?id=11910
- 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

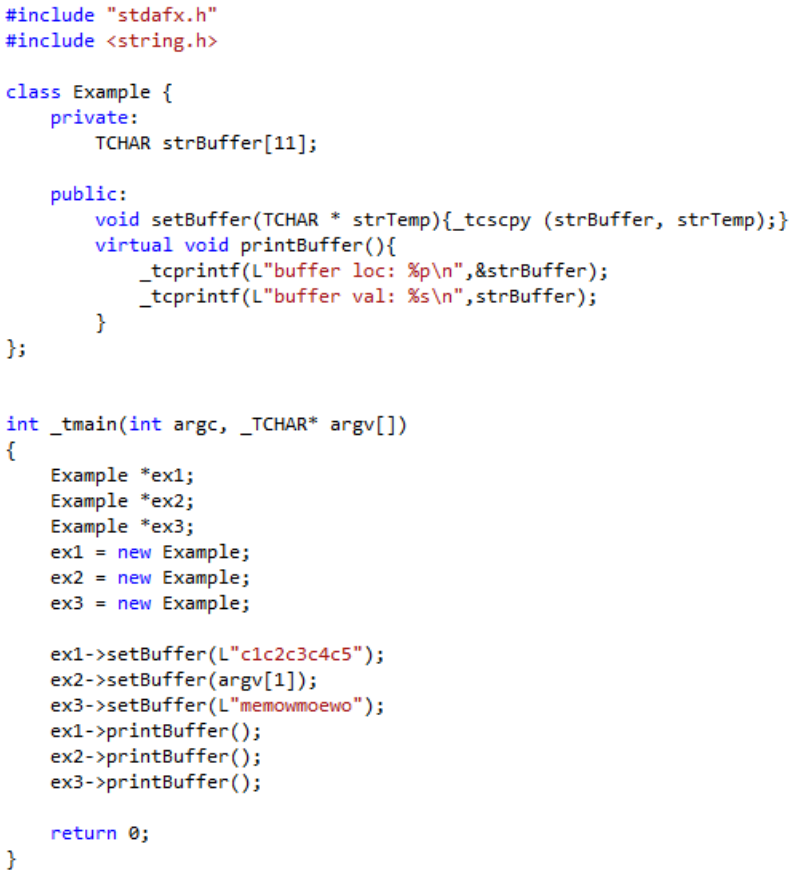
Beyond binaries

- 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…



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

class Example {
    private:
        TCHAR strBuffer[11];

    public:
        void setBuffer(TCHAR * strTemp){_tcscpy (strBuffer, strTemp);}
        virtual void printBuffer(){
            _tcprintf(L"buffer loc: %p\n",&strBuffer);
            _tcprintf(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"memowmoewo");
    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

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

http://blogs.msdn.com/b/sdl/archive/2011/12/02/security.aspx

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