.NET serialiception

.NET serialiception - agix, SCRT.

- Published: date not stated
- Original: <https://blog.scrt.ch/2016/05/12/net-serialiception/>
- Preserved from: https://blog.scrt.ch/2016/05/12/net-serialiception/ (live) on 2026-09-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Introduction

After discovering a weird base64 encoded format during pentest I wanted to find out what was that format and I met BinaryFormatter.aspx).

The BinaryFormatter format is internally used in a bunch of functions or can be used directly to materialize .NET objects.

The binary format is well documented here.

To be able to serialize and unserialize .NET objects, they must extend the ISerializable class.

As other serialization formats, the unserialized object is often casted to a specific one triggering some error if it doesn't fit.

This research focus on the one that implements some specific deserialization constructors which are executed regardless of the expected casting.

James Forshaw already did a lot of work on this subject (https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf) but firstly I wanted an easy way to tamper this BinaryFormatter format and then implement some POC everybody could use during penetration testing.

My scenario is to try to exploit this BinaryFormatter serialization from a web application. As I said, I saw it directly in POST and Cookies data during a pentest but another interesting vector is the ViewState that could pass BinaryFormatter format with the Token_BinarySerialized (even if I'm late on the subject http://fr.slideshare.net/ASF-WS/asfws-2014-slides-why-net-needs-macs-and-other-serialization-talesv20 as it's rare to find one without a HMAC).

Firstly I wrote a JSON <-> BinaryFormatter converter so it's easy to extract readable information from a BinaryFormatter stream, edit it in JSON and send back the modified version.

I quickly did the same with ViewState so I could tamper a non hmaced ViewState and insert my crafted BinaryFormatter to it.

Then I just have to find the interesting class.

ISerializable

Analysing most of the native ISerializable extended class, I noticed every actions an attacker could do by tampering a BinaryFormatter stream (I tried to be comprehensive).

Here is a list of quite interesting things we can do during deserialization :

Already known

- Initiate smb connection from the server (impacket/examples/smbserver.py could help you get the challenge to crack it with JOHN or try smbrelay)
- Delete arbitrary files (with ndp/fx/src/compmod/system/codedom/compiler/TempFiles.cs) destructor

Use examples/TempFileCollection_arbitrary_delete_or_smb_connect.json ObjectId 8 is a BinaryObjectString which contains the filepath you want to delete (if you use a UNC path it will connect to it and leak hash)

Useless at the first view

- Call any class constructor without argument ndp/clr/src/BCL/system/security/policy/hashmembershipcondition.cs

Check examples/HashMembershipCondition_arbitrary_empty_ctor_call.json ObjectId 4 contains a BinaryObjectString which contains the Assembly string of the class you want to construct. This class should have an empty argument constructor that would be called (so it seems very useless)

- Create WindowsIdentity ndp/clr/src/BCL/system/security/claims/ClaimsPrincipal.cs

I didn't try it as I didn't find an idea to exploit it.

- Call to ParseManifest (unkown) ndp/clr/src/BCL/system/activationcontext.cs

I didn't try it as I don't know what Manifest is (maybe interesting)

- Unserialize Icon, Font, Bitmap, Cursor, ImageList

ndp/fx/src/commonui/System/Drawing/Icon.cs
ndp/fx/src/commonui/System/Drawing/Advanced/Font.cs
ndp/fx/src/commonui/System/Drawing/Bitmap.cs
ndp/fx/src/winforms/Managed/System/WinForms/Cursor.cs
ndp/fx/src/winforms/Managed/System/WinForms/ImageListStreamer.cs

Could be good to exploit native parser bug remotely.

- Load X509 certificate
  ndp/clr/src/BCL/system/security/cryptography/x509certificates/x509certificate.cs

Interesting discoveries

- XXE ! ndp/fx/src/data/System/Data/DataSet.cs

DataSet object waits for XmlSchema BinaryObjectString that is vulnerable to Xml External Entities (check examples/DataSet_XXE.json).

Unfortunately Microsoft XML parser is quite susceptible so I didn't succeed to blindly leak a web.config file (I only read C:\Windows\system.ini as POC). Actually I don't know what to leak with a blind XXE on Windows System.

Maybe an XXE expert could do better, but it's still a quite critical vulnerability if you know the application (SSRF and co...)

- Unmarshal COMObject

So here comes the best part!

COMObject Marshaling

James Forshaw already pointed it ndp/fx/src/wmi/managed/System/Management/InteropClasses/WMIInterop.cs without trying to exploit it.

As COMObject Mashaling is a native windows function, I looked at the COMObject marshaling format to add a third layer of serialization (https://msdn.microsoft.com/en-us/library/cc226828.aspx) and reversed the ole32.dll CoUnmarshalInterface function.

I know nothing about COMObject so maybe I missed an easy exploit but I didn't succeed to use OBJREF_STANDARD, OBJREF_HANDLER, and OBJREF_EXTENDED.

On the other hand OBJREF_CUSTOM allows you to pass an arbirary clsid that would load the associated DLL and try to unmarshall another layer of specific data pObjectData if this COMObject exposes the IMarshal interface (kind of 4th layer of custom serialization).

Unfortunately automatically trying junk data on every CLSID I found in the register didn't produce anything.

Reversing ole32.dll shew me some special hardcoded CLSID. One of them (CLSID_StdWrapper : {00000336-0000-0000-C000-000000000046}) used with special IID (IUnknown {00000000-0000-0000-C000-000000000046}) seems to use the pObjectData and fuzzing its format gives me a crash with controlled register \o/ !

I did my first tests on specific 64bits binaries under windows 7 and then setup a windows server 2008 R2 64 bits with IIS and asp.net 2.5 but the COMobject unmarshalling from BinaryFormatter is a feature of .NET (even in the last version) and the crash occurs in ole32.dll not related to IIS version so it should work everywhere.

```
Machine Name:
Debug session time: Mon Mar 21 18:38:26.000 2016 (UTC + 2:00)
System Uptime: not available
Process Uptime: 0 days 0:00:37.000
.................................
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(1700.1d04): Access violation - code c0000005 (first/second chance not available)
*** WARNING: Unable to verify timestamp for ole32.dll
*** ERROR: Module load completed but symbols could not be loaded for ole32.dll
ole32+0xb827d:
000007fe`fde2827d 8b5f08          mov     ebx,dword ptr [rdi+8] ds:61616161`6161616
0:000> r
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000001d04
rdx=000007fefdef8ea8 rsi=6161616161616161 rdi=6161616161616161
rip=000007fefde2827d rsp=00000000001af630 rbp=0000000000000000
 r8=00000000000002aa  r9=000007fefdef8430 r10=6565656565656565
r11=00000000001af700 r12=00000000002a60d0 r13=00000000001af6f0
r14=00000000002a60d0 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202
ole32+0xb827d:
000007fe`fde2827d 8b5f08          mov     ebx,dword ptr [rdi+8] ds:61616161`6161616
```

I figured that it may be possible to control RIP from this pointer with a very specific structure.

> [Pointer + 0x20] -> [Pointer2 + 0x20] -> [Pointer3 + 0x28]
>
> -> [Pointer3 + 0x10]
>
> -> [Pointer3] == 0
>
> [Pointer + 0x38] -> [Pointer4 + 0x38] -> [Pointer5] -> [Pointer6 + 8] -> RIP

Of course finding such specific structure in randomly mapped memories was impossible.

If you remember the list of possible unserializable .NET objects one of them should raise your attention to achieve heap spray.

Bitmap ! Indeed it is possible to craft a specific BMP that would be mapped untouched in memory (see examples/Image.json). But how can we achieve heap spray without sending Gb of BMP?

BinaryFormatter allows internal object reference!

Firstly it may allow an easy DOS by cross referencing two objects (haven't tested yet).

Secondly, it's perfect to do a nice heap spray!

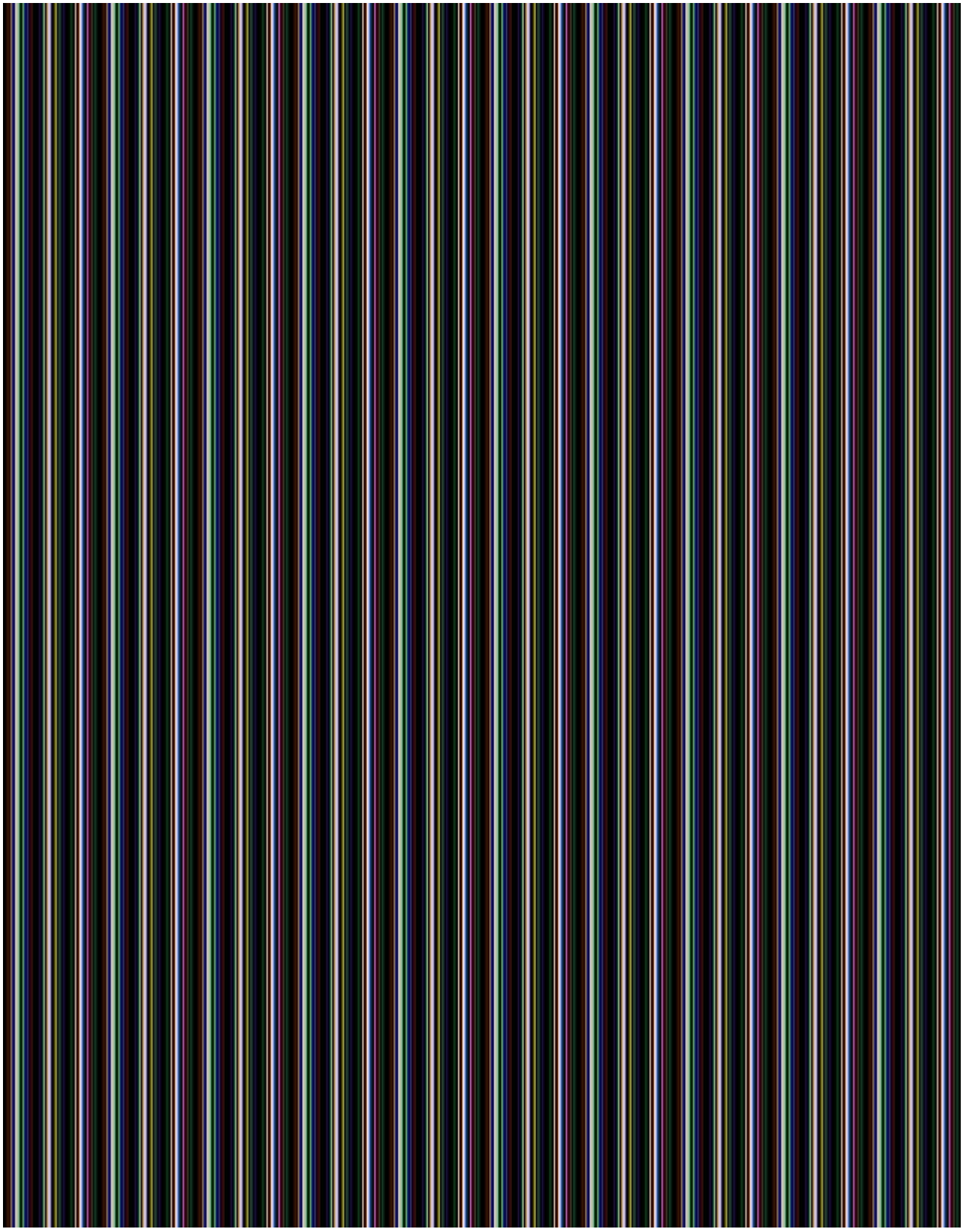
Just create one ArraySinglePrimitive with your BMP sprayed data and create as many ClassWithMembersAndTypes (System.Drawing.Image) with member "Data" a MemberReference pointing to the ObjectID of this ArraySinglePrimitive.

I choose an arbitrary bmp size aligned on 0x10000 ((640 * 818) == 0x180000 + bmp header) so it would reduce the shift in random mapping.

Once I found a nice always mapped address (0x30303030) I tried to spray the magic pointer allowing me to control RIP.

```
def q(i):
 return struct.pack('<Q', i)

spray =  q(0) + q(pointer+0x8)
spray += q(RIP-1) + q(0xDEADBABEDEFECA7E)
spray += q(pointer) + q(0xDEADBABEDEFECA7E)
spray += q(0xDEADBABEDEFECA7E) + q(pointer+0x10)
```

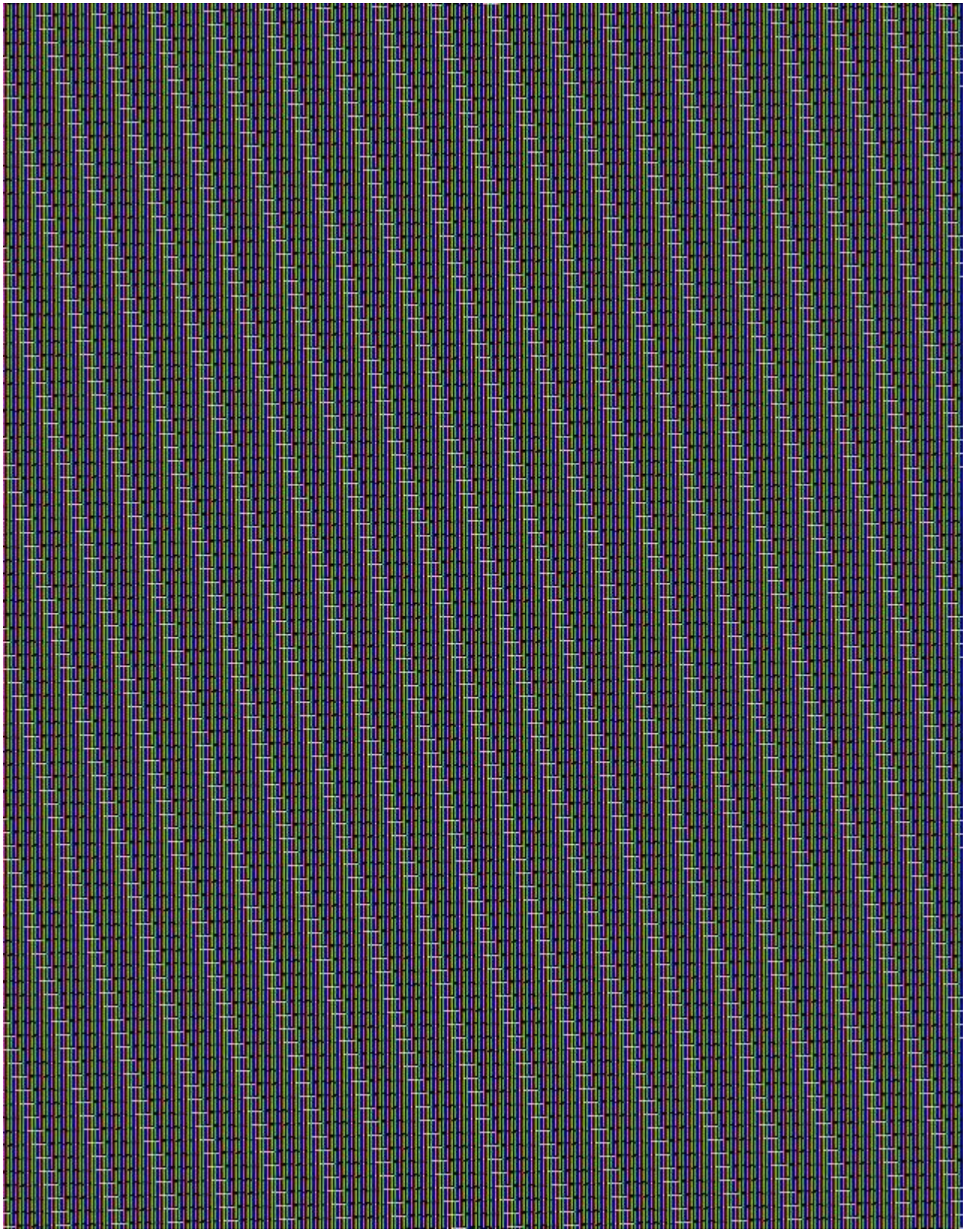


A very good point for us is that memory access errors are handled by .net so spawned IIS worker process w3wp.exe doesn't crash.

You can try as many times as you want and it should take maximum 8 tries to find your structure correctly aligned!

With relatively good RIP control, next part was to find non-ASLR DLL to construct our ROP. Fortunately, diasymreader.dll is non-ASLR and seems to be loaded after a first .net exception.

I needed to change my magic pointer structure to also control RSP with one gadget and point it in my sprayed retchain+ropchain.



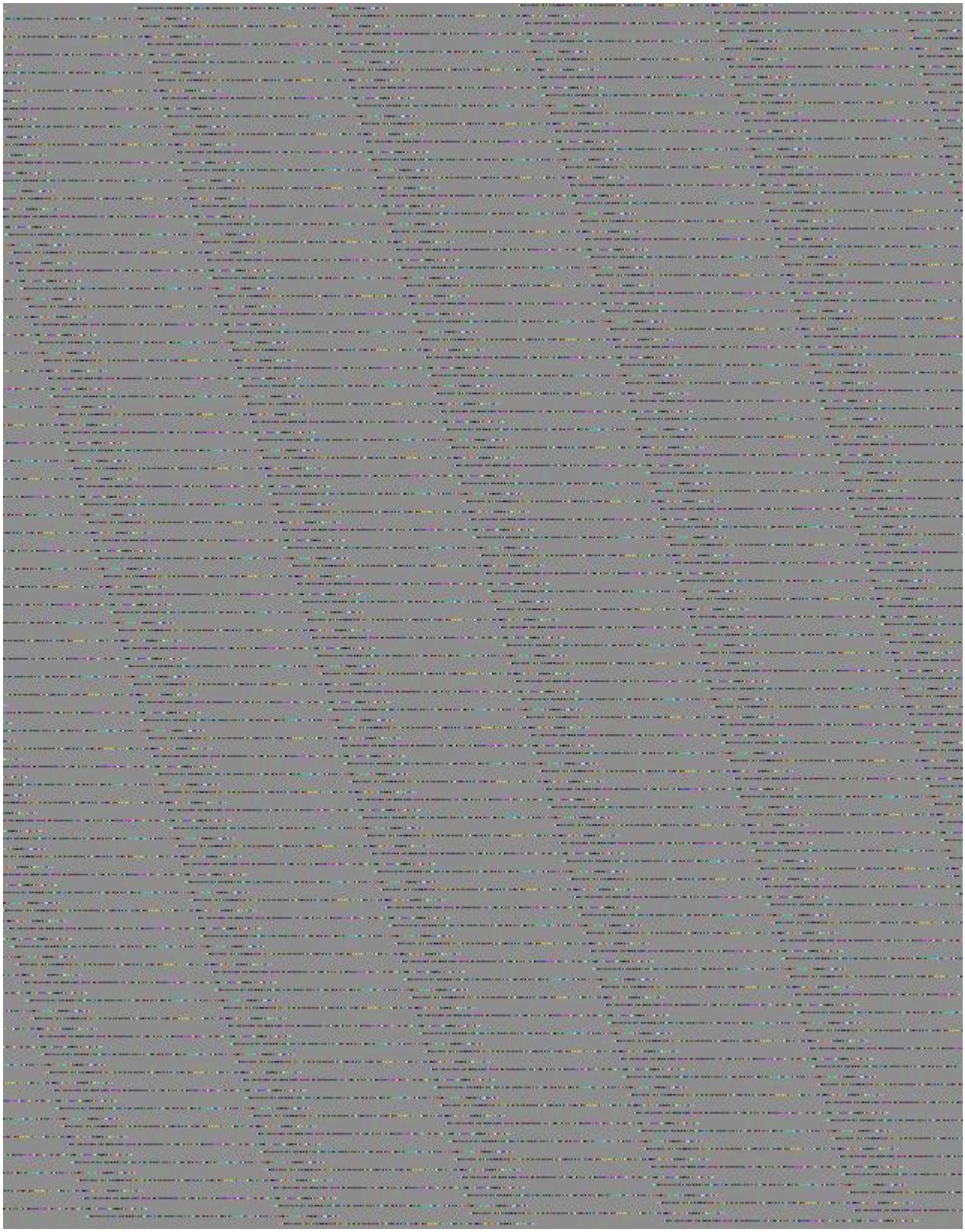
```
spray =  q(0) + q(pointer+0x8)
spray += q(RIP-1) + q(0xDEADBABEDEFECA7E)
spray += q(pointer) + q(RIP2)
spray += q(RSP) + q(pointer+0x10)
```

RIP : `push rsi # pop rsp # add rsp, 0x20 # pop rdi` RIP2 : `pop rsp # add rsp, 0x10`

This way I can control RSP !

Last part should be straight-forward as virtualAlloc and memcpy functions can be called from diasymreader.dll.

A third heapspray with nopsled+shellcode should be enough to stabilize the exploit.



I successfully reliably exploited this vulnerability on my IIS lab directly from a non hmac'ed ViewState to run meterpreter.

To sum up :

- ViewState deserialization with Token_BinarySerialized

If you directly find BinaryFormatter (with ot without obfuscation (SAP application I'm looking at you with your base64(gzip(base64(BinaryFormatter))))

- BinaryFormatter deserialization which we use to deserialize 3 BMP 400 times each and a IWbemClassObjectFreeThreaded object
- this object actually contains a malicious marshalled COMobject that allows us to control a special pointer.
- if this pointer dereference a specific structure that we spray with our first BMP we can control RIP then RSP that point on our second BMP with a ROPCHAIN that VirtualAlloc and memcpy our shellcode from the third BMP.

Conclusion

Deserializing untrusted input is bad, we all know that... I didn't find public POC of vulnerability with .NET serialization (compare to Java) so here it is!

For information, I reported the COMObject deserialization crash to Microsoft and they flagged it "*defense in depth*" as it requires untrusted unmarshaling first, so if they make a patch it will be in a long time.

Unfortunately I didn't find another vector to exploit the COMObject unmarshaling bug (RPC DCOM use another dll to unmarshal COMObject) but maybe someone with better windows knowledge could find an idea !

Last but not least the .NET remoting system internally uses BinaryFormatter. It has been replaced by WCF but it should be possible to exploit the same bugs if WCF is configured to accept NetDataContractSerializer.aspx).