View State, The unpatchable IIS forever day being actively exploited

View State, The unpatchable IIS forever day being actively exploited - Zeroed, zeroed.tech.

- Published: date not stated
- Original: <https://zeroed.tech/blog/viewstate-the-unpatchable-iis-forever-day-being-actively-exploited/>
- Preserved from: https://zeroed.tech/blog/viewstate-the-unpatchable-iis-forever-day-being-actively-exploited/ (live) on 2026-09-09
- 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.

View State, The unpatchable IIS forever day being actively exploited

Published: 21/07/2024

View state exploitation has been around for years however the complexity involved in exploiting and investigating exploitation has resulted in many defenders having little to no understanding of how to detect and more importantly, remediate a compromised host.

In this post, I'll explain how to exploit view state on a simple web application and a fully patched Microsoft Exchange 2019 host. We'll also exploit an old Microsoft Exchange post-auth CVE which requires a slightly different setup. Next, I'll outline-solid the artifacts generated by successful exploitation and how they can be used by threat hunters and responders to identify successful exploitation. Lastly, I'll cover how to remediate a network and discuss some of the challenges and dangers of doing so.

This will be a long post so grab a coffee and let's get started.

What is a View State?

View state is a mechanism used by ASP.NET web applications to maintain page state between reloads, particularly the state around controls and form elements, and is generally used as a means of working around the stateless nature of HTTP. View state is intended to maintain state whilst interacting with a single page (i.e. updating the contents of what you're currently looking at)

rather than passing data between pages. This can allow web developers to implement features such as change listeners to perform some operation when a text input value changes between form submissions. When a request is made to a server, the web application encodes the received state of controls (along with custom data) into a view state object, then serialises this object and returns it as a hidden field called `__VIEWSTATE` within the response. We can see this view state object if we view the source of the page:

[image: __VIEWSTATE in HTML source]

The next time the user submits a request to the server, the value of the `__VIEWSTATE` field will be submitted alongside the request. Before processing the user's request, the server will deserialise the `__VIEWSTATE` value back into a view state object and make it available to the processing application which can extract any control state or custom information stored there. Microsoft's page on view state) contains a solid write-up on why view state is used. It's worth noting that view state has been around since the early 2000's and many of the problems it was designed to solve have been resolved with changes in how modern web applications are developed. That being said, view state is still enabled on every single page served by ASP.NET.

View State Security

There are three levels to view state security:

- `Encrypt and Authenticate` - View states are encrypted and signed with a Machine Authentication Code (MAC), preventing end users from viewing the contents of the `__VIEWSTATE` field as well as preventing it from being tampered with
- `Authenticate Only` - View states are signed with a MAC to prevent tampering. This is the default setting on modern IIS servers
- `None` - Protecting view state with a MAC used to be opt-in. Exploiting these servers is trivial as they will deserialize any view state message sent to them

> There are a few values that determine the security level used for a given request. First, a view state will (almost) always be signed with a MAC. The `_enableViewStateMac` parameter is hard coded at the page level to `true` and IIS contains no code paths that change this value. Developers can explicitly disable validation by calling `EnableViewStateMac = false` within their application. [image: MAC enabled] Encryption is a little trickier as there are multiple paths that lead to encryption. IIS has an internal parameter called `ViewStateEncryptionMode` which can have one of three values: - `Auto` - `Always` - `Never` IIS defaults to `Auto` which causes the view state to be unencrypted unless a component of the page being loaded explicitly requests encryption (for example, a developer may create a "secure password" component and request that the view state handling this component's data be encrypted). Alternatively, a page can explicitly request its view state be encrypted by calling `ViewStateEncryptionMode = ViewStateEncryptionMode.Always`. Using either of the above methods will cause IIS to inject a hidden field called `__VIEWSTATEENCRYPTED` along side the `__VIEWSTATE` field in any page responses. This value will be sent back to the server with any future request to inform the server that the `__VIEWSTATE` value is encrypted. The last route for enabling encrypted view states is via an extra parameter on IIS machine keys which is not exposed via the UI and can only be set via the `web.config` file. `compatibilityMode` is a parameter that can be applied to the

> `machineKey` entry of a `web.config` file to change the default compatibility mode for IIS encryption from `Framework20SP1`, designed to maintain backwards compatibility with .NET 2, to `Framework45` which supports improved protection but requires .NET 4.5 or later. Checkout the Microsoft documentation for more details but to summarise, setting the `compatibilityMode` to `Framework45` will remove some older validation methods and enforce encryption for all view states. As mentioned, this value cannot be set via the UI and appears to be intended for use by application developers rather than those deploying applications. In testing I've been able to set this parameter on an auto-generated key however after doing so the `IsolateApps` parameter was no longer used, resulting in all apps using the same key. I wouldn't recommend doing this in a production environment unless you know what you're doing:

```
<machineKey compatibilityMode="Framework45" decryptionKey="AutoGenerate,IsolateApps" validationKey="AutoGer
```

> The following screenshot shows the various code paths taken based on each of the above settings [image: EncryptionModes]

If you're wondering why `authentication` is enabled by default but encryption isn't, Microsoft has the answer for us#encrypting-view-state):

> Encrypting view state data can affect the performance of your application. Therefore, do not use encryption unless you need it.

So unless the application is storing sensitive information in the view state object, there's no advantage to encrypting it.

> There's nothing to stop an adversary from sending an encrypted view state to a page which isn't expect it. By specifying `__VIEWSTATEENCRYPTED` along side their request, adversaries can force IIS to process an encrypted view state. The primary use for this is to bypass web application firewalls/other forms of packet inspection.

I've mentioned both signing and encrypting view states, both of which are operations requiring keys of some kind, be it to encrypt/decrypt the view state or to sign and later validate its contents. But where do these keys come from? It's time we talk about...

IIS Machine Keys

IIS machine keys underpin the security of several features of IIS including view state. A machine key is comprised of two keys referred to as the `Validation Key` and the `Decryption Key`, both of which can be viewed through the IIS Manager by selecting a server, site or application from the left-hand menu followed by selecting `Machine Keys` from the central panel.

[image: Key]

On a fresh install of Windows Server 2022, the Machine Key configuration looks like the image below. Let's step through each of these options so we understand what they do related to view state:

[image: IIS]

`Validation Method` determines the signing algorithm used to protect generated view state objects from tampering. We'll need to know this value when performing our attack. `Encryption Method` determines the encryption algorithm used to encrypt/decrypt view state objects. I've never encountered an application using encrypted view states but I'll briefly cover them later.

Next, we have the actual `Validation` and `Encryption` keys, both of which have been configured with key generation parameters rather than actual keys. Before we get into key generation and what these values mean, I'll cover the last option you're likely to see when inspecting Machine Keys, static keys.

[image: Keys]

Selecting `Generate Keys` on the right will disable the `Automatically generate at runtime` boxes and replace the key generation parameters with static keys as seen above. When applied, these machine keys will be saved to a `web.config` file under the web root of the selected application (`C:\inetpub\wwwroot\web.config` in my case) or `C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config\web.config` if the server level Machine keys were selected.

[image: Generated keys]

It's worth noting that as static keys are stored as a text file, they are significantly easier to steal than auto-generated keys. So why would you ever want static keys? There are situations where they have to be used, particularly when an application is distributed/running behind a load balancer. For example, if `ServerA` processes a request and returns a view state, then `ServerB` receives a follow-up request, `ServerB` will need to be configured with the same keys or it won't be able to validate the view state generated by `ServerA`.

When investigating a host configured with static keys, it's worth validating where those keys came from. There have been several instances of developers hard-coding keys into their application resulting in all deployments of said application sharing a common key. This means if an adversary gets a copy of this key by compromising a host running this application (or by downloading the application themselves), they now have the required key to attack all users of the application. One of the more notable examples of this is `CVE-2020-0688` where one of the Microsoft Exchange's web applications was configured with a static key common to all installs resulting in trivial RCE. We'll exploit `CVE-2020-0688` later in this post to show the danger of hard-coded static keys.

Another common issue is copy/paste keys. I've encountered several organisations using keys that someone (be it them or a developer) has copied from a Stack Overflow question or a Microsoft Community post. Most of these keys have been captured in the Blacklist3r project on GitHub so its always worth searching for any static keys in there to see if they are known to the world.

That's enough on static keys, let's get into

Key Generation

All web applications that aren't running as part of a distributed/load-balanced system should be using automatically generated keys, but how are they generated and where are they stored? As with all great questions, it depends.

We'll start with the generation process as that's comparatively simple. The terminology used by Microsoft in the IIS manager isn't great as `Automatically generate at runtime` implies keys will be generated *every* time IIS starts however this isn't the case. The first time IIS needs a machine key, it will generate a master machine key which will be used for the life of the server (this will become more important when we look into remediation). All other generated keys will be derived from this master key with the following modifications:

- If `IsolateApps` is specified, the master key will be modified based on the application's name
- If `IsolateByAppId` is specified, the master key will be modified based on the application's ID
- Both values can be specified

Importantly, IIS can have multiple master machine keys as not all applications have access to the storage locations of all master machine keys. For example, if `Application A` and `Application B` are configured to run as `SYSTEM`, they will both access the same master machine key however `Application C` which is configured to run as an unprivileged user will generate its own master key. We'll talk more about where these keys live in a second.

Validation keys, machine keys, master machine keys, derived keys, that's a lot of generated keys but where are they all stored? Thankfully only the master machine key is stored, all other keys are recalculated every time IIS starts so that cuts our work down significantly. As for the storage location of the master machine keys, that question is far more difficult than it should be.

On a fresh install of IIS, the two settings that determine where keys will be stored are the `identity` the application pool has been configured to run under and the state of the `load user profile` value.

[image: Identity]

How these fields affect the storage location is anything but simple, with edge cases making things extra complex so let's take a second to go through it all. The `identity` field has four possible options, `ApplicationPoolIdentity`, `LocalService`, `LocalSystem`, `NetworkService` and `Custom Account` (used when specifying explicit credentials for an application pool) whilst the `load user profile` field is a Boolean value.

Internally, IIS doesn't take these values into account when trying to find the master key for a given application pool, instead, it tries all possible locations in order of most privileged to least privileged which can cause some of the aforementioned edge cases.

First, IIS will attempt to open a handle to the Local Security Authority (LSA) so it can read an LSA secret called `L$ASP.NETAutoGenKeysV44.0.30319.0` (this value can be located in the Windows registry under `HKEY_LOCAL_MACHINE\SECURITY\Policy\Secrets\L$ASP.NETAutoGenKeysV44.0.30319.0`). As processes must be running as administrator or higher to open a handle to the LSA, this will only succeed if the application pool is running as `LocalSystem` or a `Custom Account` with administrator privileges.

Next, IIS attempts to open a handle to `HKEY_USERS\[SID]\Software\Microsoft\ASP.NET\4.0.30319.0` where `[SID]` is the SID for the current process. The success of this is based on whether the target SID has a user profile that can be loaded. In the case of `LocalService` and `NetworkService`, a user profile exists on all systems however for `ApplicationPoolIdentity` and `Custom Account`, this is dependent on if they have previously had a profile created, either by IIS being configured with `load user profile` set to `True` or a user account having been logged into in the case of `Custom Account`. Here's where we hit an edge case, if an application pool was previously configured with `load user profile` but later

changed back to not load a user profile, IIS will continue to store its keys in the user profile unless the profile is manually deleted.

If IIS still hasn't found its master key, it attempts to load a key from `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ASP.NET\4.0.30319.0\AutoGenKeys\[SID]` where `[SID]` is the SID for the current process. This is the default location keys generated for `ApplicationPoolIdentity` will be stored however for some reason, application pools configured to run as a low privileged `Custom Account` do not have permission to write to this location.

This brings us to the final option. If all else fails, IIS will generate new keys that will be used for the life of the current process however they will not be stored anywhere and will be lost when it terminates.

> Arguably, this is the most secure option :)

The table below outlines all combinations of `identity` type and `load user profile` settings and where their keys will be stored on a default install of IIS.

| Identity | Load user profile | Location | | | ApplicationPoolIdentity | False | HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ASP.NET\4.0.30319.0\AutoGenKeys\[SID] | | | ApplicationPoolIdentity | True | HKEY_USERS\[SID]\Software\Microsoft\ASP.NET\4.0.30319.0 | | | LocalSystem | True | HKEY_LOCAL_MACHINE\SECURITY\Policy\Secrets\L$ASP.NETAutoGenKeysV44.0.30319.0 | | | LocalSystem | false | HKEY_LOCAL_MACHINE\SECURITY\Policy\Secrets\L$ASP.NETAutoGenKeysV44.0.30319.0 | | | LocalService | false | HKEY_USERS\S-1-5-19\Software\Microsoft\ASP.NET\4.0.30319.0 | | | LocalService | true | HKEY_USERS\S-1-5-19\Software\Microsoft\ASP.NET\4.0.30319.0 | | | NetworkService | true | HKEY_USERS\S-1-5-20\Software\Microsoft\ASP.NET\4.0.30319.0 | | | NetworkService | false | HKEY_USERS\S-1-5-20\Software\Microsoft\ASP.NET\4.0.30319.0 | | | Custom Account (Administrator) | false | HKEY_LOCAL_MACHINE\SECURITY\Policy\Secrets\L$ASP.NETAutoGenKeysV44.0.30319.0 | | | Custom Account (Administrator) | true | HKEY_LOCAL_MACHINE\SECURITY\Policy\Secrets\L$ASP.NETAutoGenKeysV44.0.30319.0 | | | Custom Account (low priv) | false | Uses random key that is lost when the server closes Fails to write to target key | | | Custom Account (low priv) | true | HKU\[SID]\Software\Microsoft\ASP.NET\4.0.30319.0\AutoGenKeyFormat | |

Exploiting View State

Now that the background's out of the way, we can move on to how to exploit view state. My target server will be a fully patched `Windows Server 2022` instance running a basic web application. Given this is for testing purposes, I'll be skipping the initial access phase of our "compromise" and using my physical access to the target to deploy any needed webshells etc.

To successfully exploit view state, we'll need to acquire several values:

- `Validation key` The validation key component of the IIS machine key for our target application
- `Validation algorithm` The validation algorithm in use
- `Target path` The relative path to the webroot of the page we intend to target

- `App path` The path of the application we are targeting

Validation Key

To acquire the validation key, we'll utilise this handy machine key finder script https://gist.github.com/irsdl/36e78f62b98f879ba36f72ce4fda73ab which adversaries love. I can't tell you the number of times I've analysed a malicious assembly that ended up being this script wrapped into an adversaries custom payload structure. To use this script, simply drop it in a web-accessible directory then access it via a browser. Again, I'll be doing this via physical access but any form of post-compromise access should allow you to drop a file to disk. Accessing the uploaded script will produce something similar to the following:

[image: Dumped Keys]

> Depending on your setup, you may have some values in different sections. For example, the box under the first horizontal line will output the contents of keys stored under the application pool's user profiles. As we are not loading a user profile, this section is blank.

Of the values shown, we can use either:

- `initial validationKey (not useful for direct use): A28B6D6F521D332C8DF255341810995F91804D0EA011DEEB3AF79F084128C86813D6753517E2DEDC4557B9F0E336F37BF0EA13D2B5FF8D32FE61DBC6477F2C94`

or

- `App specific Validation Key (when uses IsolateApps): B298BA4E521D332C8DF255341810995F91804D0EA011DEEB3AF79F084128C86813D6753517E2DEDC4557B9F0E336F37BF0EA13D2B5FF8D32FE61DBC6477F2C94`

[image: Keys of interest]

If you look closely at these two values, you might notice that only the first four bytes are different

```
032132F70BC12AC8A5783D10805AB2FE779649749FB49C9E2302F836FD9613DFB647C5D8F04B1EF1906EE180A64D08
B298BA4E0BC12AC8A5783D10805AB2FE779649749FB49C9E2302F836FD9613DFB647C5D8F04B1EF1906EE180A64D08
```

The first value, starting with `032132F7`, is the master validation key whilst the second value is the calculated application-specific validation key which is used when an application is configured with the `IsolateApps` parameter. The mutation between the master and the application key is based on a hash of the application's name.

As for why we can use either of these values, the tool we will be using later to generate our payload can perform the same key mutation for us to calculate the application key from the master. When targeting a single application site there is very little difference between which key you use however when targeting multi-application sites like Microsoft Exchange, using the master key saves you from needing to acquire keys from each application.

Validation Algorithm

There are ways to extract this value through the use of .NET reflection but they are far more complicated than trying each of the seven possible values till one works. We can make an educated guess though as `SHA1` is by far the most common value used followed by `HMACSHA256`. `TrippleDES` and `MD5` are rarely used.

[image: Algorithms]

Target path and application path

The last two values can be derived from the URL we are targeting. `Target Path` is just the full page we want to target. This can be any ASPX page (including an empty file!) but keep in mind, that any restrictions placed on that page (e.g. authentication required) will still apply. In our case, I'll target the script I uploaded to display the machine keys so my `Target Path` will be `/keys.aspx`. The `application path` is the root path of your target application which will generally be the target path minus the resource name which in our case will be `/`. On more complex systems like Microsoft Exchange, this may be something like `/owa/`. If your exploit is failing, it may be that your target application is using a virtual directory under its application. To work around this, try dropping directories from right to left so `/test/dir` to `/test` and finally `/`.

Now that we've gathered all the values we need, it's time to exploit our server.

Exploitation

To craft our malicious view state, we'll be using YSoSerial.NET. Download the latest version from the releases tab and extract it. You may need to add an exception to your antivirus (Windows Defender nuked the download for me). Open a terminal and navigate to the `Release` directory under your extracted YSoSerial directory and run the following command:

```
.\ysoserial.exe -p ViewState -g TextFormattingRunProperties -c "COMMAND" --validationalg="VALIDATIONALGORITHM"
```

With my values, this will look like this:

```
.\ysoserial.exe -p ViewState -g TextFormattingRunProperties -c "cmd /c whoami > C:\\users\\public\\exploited.txt" --vali
```

Most of the parameters above line up with the values we've already discussed so I'll skip over those. As for the rest:

- `-p ViewState` Uses YSoSerial's view state plugin
- `TextFormattingRunProperties` Specifies the .NET gadget we want to trick IIS into deserializing. Don't worry too much about these, just know that some very smart people have found several classes that when serialised with specific values in specific fields, can trigger code execution when deserialized.

- `-c "cmd /c whoami > C:\\users\\public\\exploited.txt"` The command we want to run. Keep in mind the privileges of the target application pool when selecting a command. By default, application pool identities have very few privileges so don't expect to be writing to `C:\\Windows`.
- `--islegacy` Generates a view state using legacy keys. Legacy keys are used when `compatibilityMode` is set to a value other than `Framework45`.
- `--debug` Optional, displays some extra details about the payload generation process.
- `--validationkey` Note the `,IsolateApps` at the end of our validation key. By passing this along with the master validation key, YSoSerial will calculate the correct application key for our target. If you are using the application validation key (the second key from above) then be sure to leave this value off.

> If you are targeting a host running in `compatibilityMode` `Framework45`, the only changes you should need to make are: - Use the validation key listed under the `ASP.NET 4.5 and above` header from our key dumping script - Don't specify `,IsolateApps` - Don't pass the `--islegacy` flag Everything else should be the same.

Running this command will produce something like the screenshot below. If you passed the `,IsolateApps` parameter along with your validation key, you should get a line similar to the one in the blue box which shows the calculated application validation key. This value should be the same as the `App specific ValidationKey (when uses IsolateApps):` value from the machine key dumping script. In the red box, we have our URL and Base64 encoded payload!

[image: YSoSerial Output]

Copy the contents of your payload, then navigate to your target page in a web browser. Append a parameter of `__VIEWSTATE=` and paste in your generated payload. The final URL should look something like this.

[image: URL]

Hit enter and one of two things will happen, you'll either get a server error page or the page will load normally. If you get a server error, congratulations, you've (likely) successfully exploited view state. If the page loaded normally, IIS likely failed to validate your payload, threw a silent exception then loaded the page as normal. In my case, my payload failed because I had the wrong validation algorithm. Rerunning YSoSerial with `--validationalg="HMACSHA256"` and then rerunning my attack gave me the server error I was after.

[image: Error]

To confirm my exploitation was successful, I can verify that a text file was created at `C:\Users\Public\exploited.txt` and that it contains the value I expect:

[image: Success]

Targeting a real-world application

Let's move on to exploiting a real application. We'll be exploiting view state on an up-to-date instance of Microsoft Exchange 2019. As most of the steps are the same as in the previous sections, I'm going to focus on what's new/different. We'll start by re-dumping the IIS machine keys

by using the same script as earlier. We have to get a bit more creative when it comes to where to place our script as Exchange requires authentication to access most directories. There is however one obvious exception to this, the `/auth` directory itself! Dropping our file in `C:\Program Files\Microsoft\Exchange Server\V15\FrontEnd\HttpProxy\owa\auth` allows us to access it at `https://[HOST]/owa/auth/keys.aspx`.

[image: Exchange machine keys]

> Microsoft Exchange has been installed on the same server the prior tests were performed on. So why do we need to re-dump the keys? Shouldn't passing the `IsolateApps` flag to YSoSerial generate the right key? Nope, the master keys being used between our test app and Microsoft Exchange are different. Our original app was running as an `ApplicationPoolIdentity` which used a machine key stored in `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ASP.NET\4.0.30319.0\AutoGenKeys\[SID]` whereas Exchange is running as `LocalSystem` which loads its master key from `HKEY_LOCAL_MACHINE\SECURITY\Policy\Secrets\L$ASP.NETAutoGenKeysV44.0.30319.0`

So we have:

- `Validation Key` - `C75CE90D1F273EE3D0BB2CEADBED4C1D9FB9E1FABBBAF869BA8D58756A5223FAE27D07026FDEE86B803CFAEA988F9C02F210030164E107650F398E1AE3211E11`
- `Validation Algorithm` - `HMACSHA256` (we're just guessing here but it's likely the same as our previous target)
- `Target Path` - `/owa/auth/logon.aspx` (We could target `/owa/auth/keys.aspx` but we know that `/owa/auth/logon.aspx` will always be there on every Exchange server we target)
- `App Path` - `/owa`

> Note the `app path` is set to `/owa` not `/owa/auth`. This is because `auth` is just a virtual directory of the `owa` application rather than an application itself. [image: OWA auth]

Great, let's run YSoSerial and compromise this Exchange server. To up the complexity, we'll attack this host remotely and not rely on physical access to confirm our compromise was successful. To achieve this, we'll write our output back to the auth directory via `"whoami > C:\Program^ Files\Microsoft\Exchange^ Server\V15\FrontEnd\HttpProxy\owa\auth\out.txt"` which should allow us to access it via a web browser.

[image: Exploit]

We launch our attack by passing our payload to `/owa/auth/logon.aspx?\_\_VIEWSTATE=[PAYLOAD]`

[image: Success]

While that may look different to what we got last time, it's actually what we're after! Last time we were attacking from the local machine which caused IIS to output a detailed error message for debugging purposes, now that we're attacking remotely, we're seeing Exchange's custom error page instead.

We can confirm our attack worked by visiting `https://[TARGET]/owa/auth/out.txt`

[image: Output]

Beautiful, we exploited the server and can now run commands as `nt authority\system`!

CVE-2020-0688

The last exploitation topic I want to cover is `CVE-2020-0688`. `CVE-2020-0688` is an RCE vulnerability in Microsoft Exchange 2016 caused by the presence of a static IIS machine key on one of Exchange's backend applications. For a detailed write-up on this vulnerability checkout the Zero Day Initiative's post.

Exploiting this vulnerability is 90% the same as our previous attempts however this time all of the values we previously had to acquire are static to all Exchange 2016 servers vulnerable to `CVE-2020-0688`:

- `Validation Key` - `CB2721ABDAF8E9DC516D621D8B8BF13A2C9E8689A25303BF`
- `Validation Algorithm` - `SHA1`
- `Target Path` - `/ecp/default.aspx`
- `App Path` - `/ecp`

That everything right? Are we ready to attack? Not quite. Unfortunately, the entire `/ecp` route requires authentication, meaning attempts to access any resource behind `/ecp` will result in a redirection back to `/owa/auth/logon.aspx`. To exploit this vulnerability we're going to need to be authenticated.

> If you're wondering why we can't just target `/owa/auth/logon.aspx` like we did last time, on this version of Microsoft Exchange 2016 the machine keys are different between the `/owa` and the `/ecp` applications. If we look at the machine key configuration for the `/owa` application we can see it is set to use autogenerated keys [image: OWA] Whereas the `/ecp` application has been configured with the static keys that cause the vulnerability we are targeting [image: ECP] Because of this, the machine keys we have access to are only useful for targeting pages under the `/ecp` application. If you were to go and dump the IIS master machine key like we did in the previous section then you could target any application pool using automatically generated keys.

So we "acquire" credentials from somewhere and log into the Exchange Control Panel application, now are we ready? Almost, however now that we're logged in we need to grab two more values, the `ViewStateUserKey` and the `ViewStateGenerator`

ViewStateUserKey is an optional feature of view state which allows applications to specify additional data that goes into the validation of a view state. Typically this value will be some form of unique user identifier that cannot be determined pre-authentication. In the case of Microsoft Exchange 2016, the value used for the `ViewStateUserKey` is the contents of the `ASP.NET_SessionId` cookie, a value that is only set post-authentication. To grab this value, open the developer tools in your browser (generally `F12`) then navigate to:

- `Chromium-based browsers` Application > Cookies
- `Firefox` Storage > Cookies

and copy the value of the `ASP.NET_SessionId` cookie (`4805c5a3-54c7-4c15-8407-41907efd9256` in my case).

[image: Cookie]

To acquire the last value we'll need, simply view the page's source, search for `__VIEWSTATEGENERATOR` and copy its value. In the case of Exchange 2016, I believe this value is always `B97B4E27`

[image: Generator]

We are finally ready to launch our attack. Here are all the values we'll need:

- `Validation Key` - `CB2721ABDAF8E9DC516D621D8B8BF13A2C9E8689A25303BF`
- `Validation Algorithm` - `SHA1`
- `Target Path` - `/ecp/default.aspx`
- `App Path` - `/ecp`
- `Viewstate User Key` `4805c5a3-54c7-4c15-8407-41907efd9256`
- `Viewstate Generator` `B97B4E27`

We don't need the target and app paths anymore as the view state generator value contains the equivalent information.

> If you want to see this in action, try running the following command

```
.\ysoserial.exe -p ViewState -g TextFormattingRunProperties -c "dir" --validationalg="SHA1" --path="/ecp/default.aspx" -
```

> The debug output should include the line `Calculated __VIEWSTATEGENERATOR (ignored): B97B4E27`, the same view state generator value we extracted from the target page.

Let's generate our payload. Most of the values in the screenshot below should look fairly familiar by now. The only changes worth noting are the addition of the `--generator` and `--viewstateuserkey` parameters to incorporate our new values, the removal of the `--path` and `--apppath` parameters which have been replaced by the view state generator and the removal of the `IsolateApps` parameter from our validation key as we're targeting a static key.

[image: Payload]

With our payload generated, we are *finally* ready to launch our attack! We can fire off the payload at `/ecp/default.aspx`, and receive our expected error page

[image: Error]

And confirm our exploitation by checking our output file

[image: Output]

> The original Zero Day Initiative article references having to pass `__VIEWSTATEGENERATOR=B97B4E27` along with `__VIEWSTATE=[PAYLOAD]` in our attack URL however in my testing the exploit worked with or without specifying the generator value.

Detecting View State exploitation

We've gone through several exploitation scenarios including exploiting a basic test application, an enterprise-grade email server and a well-known CVE. Now let's turn our attention to how someone

might detect these compromises. Our main source of view state detection is good old Windows Event logs, specifically the `Application` logs. The event (singular) we are interested in is event ID `1316` from the `ASP.NET 4.0.30319.0` source. This event provides us with just about all the information we could want including:

- The time of the request
- The targeted application pool
- The targeted page
- The request URL
- The attackers IP (or more likely the load balancer's IP)
- The payload!

> The fact the event exists means an invalid view state passed validation meaning it's almost guaranteed to be malicious

Below is an event from when we compromised our Exchange 2019 server earlier:

[image: Event]

From this we can see that a malformed (but valid) view state was processed at `13/07/2024 11:31:04 PM` by the `/owa` application, by sending a request to `https://192.168.189.130:443/owa/auth/keys.aspx` with the parameter

```
/wEy7wcAAQAAAP////8BAAAAAAAAAAwCAAAAXk1pY3Jvc29mdC5Qb3dlclNoZWxsLkVkaXRvciwgVmVyc2lvbj0zLjAuMC4wLCBDd
```

> Keep in mind, `__VIEWSTATE` can (and often is) be sent via a `POST` request. Thankfully the logs capture the same information (although the URL will be a little shorter) [image: Logs]

Awesome, so we know someone attacked our server, but what did they do? I don't want to get too deep into how to fully process view state payloads at this stage so for today we'll just Base64 decode the value under `PersistedState` in `CyberChef` and eyeball it.

[image: CC]

It's a little messy but still easy enough to read with a call to CMD visible in the middle.

Other detection options

Unfortunately, there aren't many. I have had some luck with extracting view state payloads from memory dumps of IIS worker processes however you tend to have to get a memory dump extremely close to the adversary being live or have an exceptionally quiet server.

Machine key registry artifacts

There's not a huge amount available to us aside from the key itself however we can get the time the key was generated. This value is in slightly different places/formats depending on whether it's stored as an LSA secret or a regular key

Machine Key Timestamps (LSA secret)

Machine keys stored under an LSA secret store their creation time in the following key:

```
HKLM\SECURITY\Policy\Secrets\L$ASP.NETAutoGenKeysV44.0.30319.0\CupdTime\(Default)
```

This value contains the byte representation of an LDAP/FileTime timestamp which can be extracted with the following PowerShell script:

```
[datetime]::FromFileTimeUtc([System.BitConverter]::ToInt64([byte[]](Get-ItemPropertyValue -Path "HKLM:SECURITYPol
```

Machine Key Timestamps (Registry)

Machine keys stored under a regular registry key have a value called `AutoGenKeyCreationTime`.

[image: Key Time]

This value contains an LDAP/FileTime timestamp which can be parsed with sites like https://www.epochconverter.com/ldap or via the following Powershell command:

```
[datetime]::FromFileTimeUtc(133655977478251881)
```

Remediation

So you've gone through your event logs and found evidence of your server processing a malicious view state, what do you do? After exploiting the vulnerability ourselves, we know that the adversary must have access to the `Validation Key`, `Validation Algorithm`, `Target Path`, `App Path` and possibly the `Viewstate User Key` and `Viewstate Generator`. Changing the `Target Path` or `App Path` isn't a good option as the adversary can trivially find the new values. Likewise for the `ViewState Generator` which is based on the two previous values. We could change the `Validation Algorithm` but given how few options there are it wouldn't take the adversary long to guess the new value. If they targeted a page requiring authentication then resetting the compromised user's credentials is a good idea but won't keep them out for long as all they need to do is target the next user with weak credentials. Our only real option is to reset the IIS machine keys. So let's open the IIS manager, select our site, open the `Machine Key` page and click the reset key butt...

[image: Keys]

...shit.

So Microsoft doesn't provide a mechanism to reset the IIS master machine keys. That leaves us with a few options depending on our scenario.

WARNING

IIS machine keys aren't just used for view state, depending on the application they may also be used as encryption keys meaning if we forcefully reset them, we may cause irreparable

damage/data loss. Before attempting a machine key reset, be sure you have a full backup and/or a snapshot of your server. I have reset the machine keys of Microsoft Exchange servers with no *noticeable* issues however when I've reset the machine keys of a Microsoft SharePoint server it rendered it inoperable. If in doubt, contact the application's developer before undertaking a reset.

If the compromised web app was using static machine keys, simply click the `Generate Keys` button on the right of the Machine Key UI in IIS manager to regenerate the compromised keys. If the static key was shared across multiple load-balanced hosts, be sure to copy the newly generated key to all hosts.

If the compromised web app was using automatically generated keys, you can move to using static keys by selecting the `Generate Keys` button on the right. Just remember that this will write your machine keys to a `web.config` file that can be accessed via SMB/webshells etc. I wouldn't recommend this option but it is a quick fix if you are actively under attack.

Next, there's the nuclear option, rebuild the host from scratch then restore any user data from a backup. **IMPORTANTLY** do not restore the entire host from backup/restore anything to do with the registry, there's a fair chance you'll end up restoring the compromised keys onto the new host.

Lastly, if you want to repair the current host but still use automatically generated keys, it's time to follow

The Zeroed.tech IIS machine key reset procedure (don't sue me if something breaks)

I've spent hours reverse engineering, debugging and monitoring IIS servers and have come up with the following procedure for resetting IIS machine keys. Again, make sure you have a backup/snapshot before performing the following steps.

First up, we need to identify where the machine key for each application pool is located. This entails checking all of the locations I identified in the Key Generation section for the presence of a key. This can get very tedious, especially if you have multiple application pools so I've developed a script for identifying the location and generation time of all machine keys on a host. You can grab a copy from my GitHub repo but as with anything on the internet, I recommend you read through and understand this script before running it on a production host. The script will need to be run as `SYSTEM` if any of your application pools are running with elevated privileges.

[image: Script]

The output of the script will include the application pool name, user name and sid used for the application pool, and the date any machine keys were generated. Now that you have the path to all of the machine keys that need to be reset, all that's left to do is to fire up regedit (as `SYSTEM`) and delete all of the listed keys.

Again, this can be very tedious so I've written a script to automate the process which can be found here.

> This script recursively deletes keys with SYSTEM privilege. Whilst every care has been taken to test this script, I highly recommend reviewing its contents yourself and ensuring you have adequate backups before executing it. If you'd prefer to do things manually, you'll need to

> delete the following keys for each application pool SID: - `HKLM\SECURITY\Policy\Secrets\L$ASP.NETAutoGenKeysV44.0.30319.0` to reset the IIS machine key for all privileged application pools - `HKU\[SID]\Software\Microsoft\ASP.NET\4.0.30319.0` to reset the IIS machine key for each application pool linked to a profile - `HKLM\SOFTWARE\Microsoft\ASP.NET\4.0.30319.0\AutoGenKeys\[SID]` to reset the IIS machine key for each application

Running the above script with `SYSTEM` privileges will delete all auto-generated IIS machine keys on your host and produce output similar to the following:

[image: Script]

Note that only the first `MSExchange` related application pool's key shows as deleted. You'll notice in the previous screenshot which displayed the location of all machine keys that all `MSExchange` related application pools refer to the same master IIS machine key so by deleting it for one, we've deleted it for all.

> If your system has gone through some configuration changes/has been running for years, it may be worth running the reset script up to three times to ensure all keys are gone. I have seen situations where running the script once removes a key from the user profile, only to have IIS default back to a key stored under HKLM. See the edge cases section earlier in the post for more details on how this can happen.

We've deleted the old keys but IIS will still have them in memory, leaving it vulnerable to attack. To complete the reset procedure, we can restart the IIS service through the IIS manager, by running `iisrestart` or by rebooting the host. If possible, I recommend rebooting the host as that will ensure everything fires up cleanly after poking around and deleting things from the registry (and ensuring anything that may have been lurking in memory has been purged).

> If you immediately run the auditing script again you may notice that some/all of your application pools no longer have IIS machine keys. Don't worry, IIS will regenerate any keys when they are first needed. If you want to verify this, browse to a page hosted on your web server then rerun the auditing script and you should see new keys have been generated.

And that's it! You've successfully (hopefully) reset your IIS machine keys.

Wrapping up

If you've made it this far, well done. This is a complex topic that can be very hard to understand until you've dug in and had a play. In this post, we only scraped the surface of what view state exploitation can do. When adversaries use this technique, they aren't echoing command output into a text file, they are writing webshells, reflectively loading ghost shells and other assemblies, disabling system security etc. I highly recommend searching for signs of compromise in your Windows event logs, especially following a host reboot. You never know, you might just find someone trying to regain access.