

# Google App Engine Java security sandbox bypasses

Technical Report

Ver. 1.0.0

*SE-2014-02 Project*

## DISCLAIMER

INFORMATION PROVIDED IN THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW NEITHER SECURITY EXPLORATIONS, ITS LICENSORS OR AFFILIATES, NOR THE COPYRIGHT HOLDERS MAKE ANY REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR THAT THE INFORMATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS, OR OTHER RIGHTS. THERE IS NO WARRANTY BY SECURITY EXPLORATIONS OR BY ANY OTHER PARTY THAT THE INFORMATION CONTAINED IN THE THIS DOCUMENT WILL MEET YOUR REQUIREMENTS OR THAT IT WILL BE ERROR-FREE.

YOU ASSUME ALL RESPONSIBILITY AND RISK FOR THE SELECTION AND USE OF THE INFORMATION TO ACHIEVE YOUR INTENDED RESULTS AND FOR THE INSTALLATION, USE, AND RESULTS OBTAINED FROM IT.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL SECURITY EXPLORATIONS, ITS EMPLOYEES OR LICENSORS OR AFFILIATES BE LIABLE FOR ANY LOST PROFITS, REVENUE, SALES, DATA, OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, PROPERTY DAMAGE, PERSONAL INJURY, INTERRUPTION OF BUSINESS, LOSS OF BUSINESS INFORMATION, OR FOR ANY SPECIAL, DIRECT, INDIRECT, INCIDENTAL, ECONOMIC, COVER, PUNITIVE, SPECIAL, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND WHETHER ARISING UNDER CONTRACT, TORT, NEGLIGENCE, OR OTHER THEORY OF LIABILITY ARISING OUT OF THE USE OF OR INABILITY TO USE THE INFORMATION CONTAINED IN THIS DOCUMENT, EVEN IF SECURITY EXPLORATIONS OR ITS LICENSORS OR AFFILIATES ARE ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS.

INTRODUCTION.....	6
1 GOOGLE APP ENGINE FOR JAVA - SECURITY ARCHITECTURE .....	8
1.1 BASE CONCEPTS .....	8
1.1.1 File System access .....	8
1.1.2 Network access .....	8
1.1.3 Threads.....	9
1.1.4 java.lang.System class.....	9
1.1.5 Class loaders.....	9
1.1.6 Reflection API.....	10
1.1.7 JRE Class White List.....	10
1.2 JAVA SANDBOX IMPLEMENTATION .....	10
1.2.1 Communication channels.....	11
1.2.2 Virtual File System.....	11
1.2.3 Socket proxy.....	12
1.2.4 Class Sweeper .....	12
1.2.4.1 Class Loader PreVerifier .....	13
1.2.4.2 Finalizer Visitor.....	15
1.2.4.3 API Interjection and Interception .....	15
1.2.5 Class stubs .....	19
1.2.6 Class Loader architecture.....	19
2 VULNERABILITIES .....	21
2.1 METHODOLOGY USED.....	21
2.2 DETAILS .....	21
2.2.1 Issues 1, 2, 4 and 6 .....	21
2.2.2 Issue 3 .....	23
2.2.3 Issue 5 .....	24
2.2.4 Issue 7 .....	25
2.2.5 Issues 8 and 10.....	25
2.2.6 Issues 9, 11, 15 and 16.....	26
2.2.7 Issue 12 and 14 .....	27
2.2.8 Issue 13 .....	29
2.2.9 Issues 17 an 18.....	29
2.2.10 Issue 19 .....	30
2.2.11 Issue 20 .....	30

2.2.12	Issue 21 .....	31
2.2.13	Issues 22, 23, 25, 26 and 27 .....	31
2.2.14	Issue 24 .....	32
2.2.15	Issues 28 and 29 .....	33
2.2.16	Issue 30 .....	35
2.2.17	Issue 31 .....	35
2.3	AFFECTED COMPONENTS .....	35
2.4	VENDOR'S EVALUATION.....	36
2.4.1	WAI issues .....	37
2.4.1.1	Additional arguments.....	37
2.4.1.2	Closing thoughts.....	40
3	EXPLOITATION TECHNIQUES.....	41
3.1	SPECIFIC EXPLOITATION VECTORS .....	41
3.1.1	Generic privilege elevation scenarios .....	41
3.1.2	URLClassLoader instance .....	42
3.1.2.1	sun.swing.AccessibleMethod (Oct 2012 exploit vector).....	43
3.1.2.2	sun.swing.SwingLazyValue (Oct 2014 exploit vector).....	44
3.1.3	invoke() in a privileged CL namespace.....	45
3.1.3.1	Abuse of an outer class implementation (Issue 5).....	46
3.1.3.2	MethodHandleProxies implementing PrivilegedAction interface (Issues 23-27)...	47
3.2	EXPLOIT CHAINS.....	51
3.3	NATIVE CODE EXECUTION.....	52
3.3.1	Breaking type safety.....	52
3.3.2	Breaking memory safety .....	52
3.3.3	Gaining code execution.....	53
3.3.3.1	methodOop's adapter handle .....	53
3.3.3.2	NativeSignalHandler.....	54
3.3.3.3	Generic code_handle .....	54
3.3.3.4	Native code execution setup .....	55
3.3.4	Native API.....	56
3.3.4.1	Arbitrary memory access .....	57
3.3.4.2	Native code execution .....	57
3.3.4.3	Symbol address lookup .....	57
3.3.4.4	Malloc and free primitives .....	57

3.3.4.5	Sample uses.....	58
4	VULNERABILITIES IMPACT.....	59
5	SUMMARY.....	60
	REFERENCES.....	61
	APPENDIX A.....	63
	APPENDIX B.....	66

## INTRODUCTION

In Sep 2012, when our Java SE security research [1] was deemed to be complete<sup>1</sup>, we started to wonder whether any of the issues spotted in JVM implementations coming from Oracle and IBM could affect the code of other major software vendors. We also wanted to verify whether security and privacy of users' data and applications is properly implemented in the environment of an arbitrary cloud service based on a Java VM runtime. This is how we ended up investigating security of Google App Engine for Java (GAE) [2], a platform as a service (PaaS) cloud computing platform from Google that allow for arbitrary Java applications development and hosting in the company's managed data centers. This is also how SE-2014-02 project was born.

Our work on the project was started in Oct 2012. This was also the time when our initial complete GAE Java security sandbox escape was achieved<sup>2</sup>. Due to the fact that we conducted 3 other non-commercial research projects in the meantime<sup>3</sup>, our GAE work needed to be postponed several times. We finally came back to the project in Oct 2014.

On Dec 6, 2014, as a result of our a little bit more aggressive poking<sup>4</sup> around the OS sandbox underlying the GAE JVM layer, Google suspended our test account. Instead of playing a catch and mouse game<sup>5</sup> with Google, we decided to inform the public about the existence of our GAE project and reveal some brief information about the results obtained so far [3]. Taking into account an educational nature of the security issues found in GAE Java security sandbox and what seemed to be an appreciation Google had for all sorts of sandbox escapes [4], we expressed our hope the company would make it possible for us to complete the project and reenale our suspended account.

On Dec 7, 2014 as a response to Google's interest to "get whatever information Security Explorations had on the vulnerabilities" and regardless of implying that "a couple of more days to work on the project were needed in order to bring it up to the usual quality when reporting issues to vendors", information regarding vulnerabilities and associated Proof of Concept (POC) codes were sent to the company.

Google has been able to reproduce reported issues locally<sup>6</sup>, but when tried in production some of them didn't seem to work. On Dec 11, 2014, Google said that it would be OK for the company that we continue the research as long as it is done within the Java VM and not moved on to the next sandboxing layer (OS sandbox). We agreed to Google proposal and

---

<sup>1</sup> Issue 50 was supposed to mark an end to our Java SE security research project.

<sup>2</sup> In our Devovx 2012 presentation (slide 56), the big SW vendor mentioned is Google.

<sup>3</sup> Java SE (SE-2012-01), Oracle Java Cloud service (SE-2013-01) and Oracle Database Java VM (SE-2014-01) security research projects.

<sup>4</sup> We issued various system calls / intentionally triggered certain program faults in order to learn more about the nature of the error codes associated with a process death.

<sup>5</sup> We could setup another account from a different IP address, modify the POC codes, withhold from interfering with the OS sandbox, etc.

<sup>6</sup> Many of our POC's were developed in a local GAE environment, which aimed to emulate Google production environment. Unfortunately, our custom local GAE environment didn't properly mirror the Google App Engine class loading behavior (many classes marked as vulnerable were not immediately available to user code in production GAE). More on that in paragraph 1.2.4.3.

informed the company that our research will be continued with a scope limited to GAE Java VM layer.

Over the following four days we were able to confirm 21 initial issues in a GAE production environment. By the mid of Jan 2015, additional 10 issues were confirmed and reported to Google.

This paper presents the results of our research into the security of a Java security sandbox of Google App Engine. While it omits technical details pertaining to the OS sandbox layer<sup>7</sup>, we believe that the published material still constitutes a valuable source of information for all parties interested in a security of Java and Java cloud based solutions such as PaaS.

The goal of this paper is to educate users, developers and possibly vendors about security risks associated with certain design and architecture choices for cloud environments based on JRE. The other goal is to show the very tricky nature of Java security and especially the pitfalls one can easily get into if custom Java Runtime modifications are applied to certain security sensitive Java APIs and components.

In the first part of this paper, quick introduction to Google App Engine for Java security architecture is made. It is followed by a brief description of key components comprising GAE sandbox implementation. Java API interception, Class Loaders architecture and JRE Class Whitelisting are explained as part of it.

The second part of the paper presents vulnerabilities found during SE-2014-02 project. We show how single and quite innocent looking GAE Java security breaches can lead to serious, full-blown compromises of GAE / JRE security sandbox. Technical details and exploitation techniques for the vulnerabilities found during SE-2014-02 research project are also presented.

The paper wraps up with a brief information regarding vulnerabilities impact followed by a few summary words regarding Google's approach to securing Java Runtime in a cloud based environment.

Throughout this paper, whenever a reference to a GAE environment is made, GAE for Java is implied. Similarly, the term "GAE [security] sandbox" implies the GAE Java VM security sandbox, not the OS sandbox.

---

<sup>7</sup> Per agreement with Google.

# 1 GOOGLE APP ENGINE FOR JAVA - SECURITY ARCHITECTURE

Google App Engine for Java makes it possible to host and run user web applications on a Google managed server infrastructure. For security reasons, these applications are executed in a sandboxed environment. The sandbox itself is comprised of two layers. The first layer is a GAE Java sandbox, which is built on top of the underlying Java SE software. The second layer is a native OS sandbox, which limits the exposure of the operating system to user applications and a GAE environment itself.

That approach seemed to be quite natural. Back in 2012, Java didn't constitute a strong security posture due to multiple security vulnerabilities being discovered in it and several incidents involving 0-day attack codes spotted in the wild [5][6]. User code execution on a bare JRE stack was not even possible in Oracle's own Java Cloud service environment<sup>8</sup>. The less surprising it was that Google decided to implement an additional security layer on top of Java SE.

## 1.1 BASE CONCEPTS

The presence of standard Java SE security sandbox along with a GAE sandboxed layer enforced several restrictions to user applications executed in GAE environment [7]. Below, a brief characteristic is provided with respect to the key concepts that shaped the security of a GAE environment.

### 1.1.1 File System access

By default, user applications cannot access the file system of the underlying native OS. User code has only read access to its unique application deployment directory and all of its subdirectories. This means, that both user classes, resources and JSP files that make up a target application container (WAR file) can be freely read. The actual access can be implemented by the means of Class Loader's functionality (class or resource loading) or with the use of standard Java APIs (`File` / `FileInputStream` based classes).

User applications cannot write any data to the file system. They need to use the App Engine Datastore API for any persistent data storage.

### 1.1.2 Network access

GAE applications can access network resources with the use of network sockets (Java sockets API), but there are some restrictions enforced on the way they can be used. This in particular includes the following:

- sockets are available to paid apps only (as of Oct 2014)<sup>9</sup>,
- only outbound, client (non-listening) TCP or UDP sockets can be created,
- sockets cannot be bound to a specific address or port,

---

<sup>8</sup> Applications deployed by users in a target WebLogic server instance of Oracle Java Cloud Service were subject to the verification (and translation) aimed to disallow access to forbidden (potentially insecure) classes and / or functionality.

<sup>9</sup> Our tests conducted prior to that time indicate that sockets were available to all apps.

- private Google IP ranges are blocked with the exception of a few predefined hosts (DNS, SMTP, POP3S and IMAPS servers).

User applications can also make use of `java.net.URL` to open arbitrary HTTP and HTTPS connections. However, the implementation of protocol handlers for these protocols relies on the URL Fetch API [8], not network sockets.

### **1.1.3 Threads**

User applications are web applications and as such they handle web requests through Servlet API or JSP files. User code is usually spawned for the time of an associated HTTP request. GAE makes sure that all threads associated with a given HTTP request are terminated upon finishing of a request processing. This includes both successful and erroneous requests. GAE code tries to detect deadlocked threads as well. There are also both soft and hard time limits implemented of which goal is to terminate the execution of any user thread that does not end the processing within the predefined time limits.

The idea behind all of the above is twofold. GAE makes sure that no user threads outlive the HTTP requests that triggered their creation. But, this even goes further as GAE makes sure that no user code gets executed after HTTP request has been handled. This includes all sorts of system Java handlers and finalizers in particular.

While support for the so called background threads and cron tasks is available in GAE, they are not implemented as classic background Java threads (threads created with the use of `new Thread()` call, detached from the application thread group and marked as background in JRE).

### **1.1.4 `java.lang.System` class**

Most of `java.lang.System` API is not available to user applications. The `exit()` and Garbage Collector related methods such as `gc()`, `runFinalization()` and `runFinalizersOnExit()` do nothing in App Engine. As a result, user code cannot implicitly trigger the GC operation.

Methods that implement arbitrary library loading such as `load()` or `loadLibrary()` always raise a `SecurityException`. The same applies to the `setSecurityManager()` method.

### **1.1.5 Class loaders**

GAE makes it possible for user code to create arbitrary Class Loaders. As a result, user applications can successfully define and create instances of subclasses of `java.lang.ClassLoader` class that implement a custom class loading logic. The permissions of the classes defined by such Class Loader objects are enforced to be always reflecting the allowed set of permissions for user applications (permissions of an unprivileged web application). The latter enforcement is in particular important as Class Loaders can provide the JVM with classes definitions as well as their privileges (Protection Domains and permissions).

### **1.1.6 Reflection API**

GAE allows for full and unrestricted Reflection API access to application's own classes (classes defined by an application Class Loader or a custom, user defined Class Loader). User applications can reflect on private members of classes. They can also call `setAccessible()` method on them. This makes it possible to override standard Java protection mechanisms and access private class members (call private methods, read and set private fields).

An application is also allowed to reflect on JRE and API classes, but it can only access public members of these classes.

An application cannot reflect against any other classes not belonging to itself. It cannot use the `setAccessible()` method to circumvent these restrictions.

### **1.1.7 JRE Class White List**

In order to minimize the risks posed by a security vulnerability present in JRE, GAE employs the idea of JRE Class White List. Its goal is to limit the set of JRE classes that can be accessed by user code. Arbitrary class loading or linking is successful only if a requested class is allowed in the environment (it is present on a list of deemed to be safe JRE classes - the JRE Class White List).

As of Oct 2014, the JRE Class White List contained 1650+ classes and the JRE was based on Java SE 7 class base.

## **1.2 JAVA SANDBOX IMPLEMENTATION**

GAE environment contains support for all of the abovementioned concepts in order to implement a security sandbox for user applications.

GAE Java Runtime sandbox is implemented at both native and Java class level. It's building blocks are illustrated on Fig. 1. Below, a more detailed information is provided with respect to this sandbox implementation.

## GAE JVM sandbox

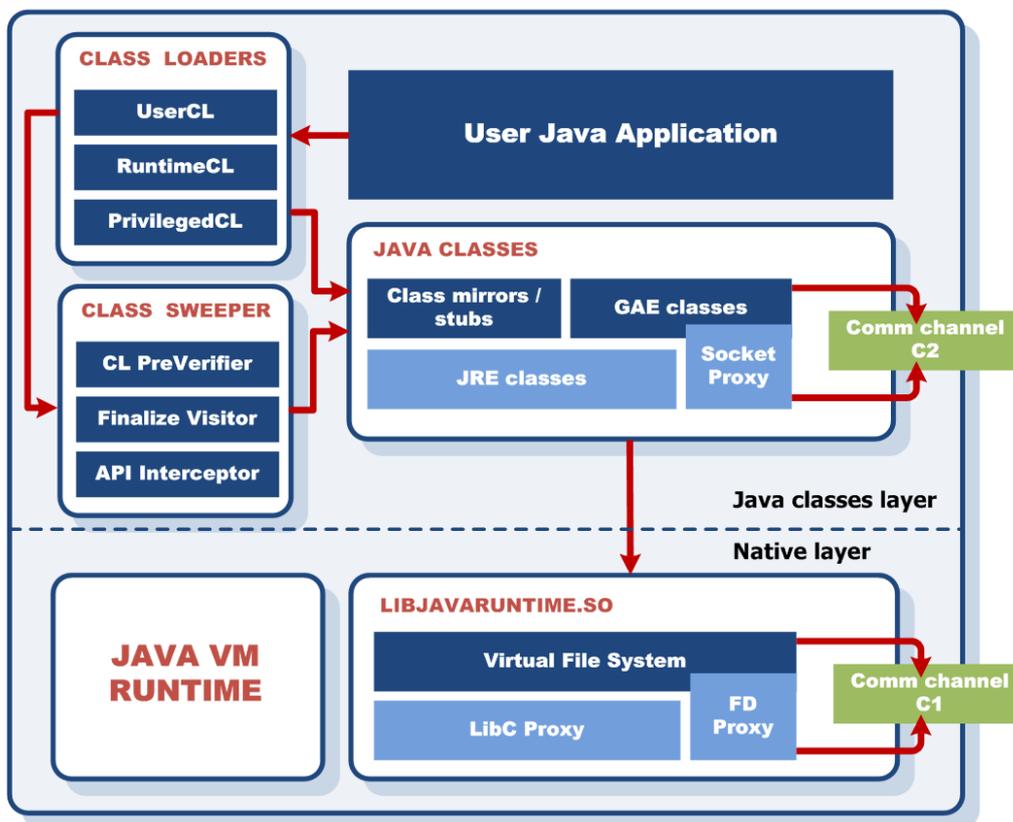


Fig. 1 The building blocks of a GAE Java Runtime sandbox.

### 1.2.1 Communication channels

GAE Java Runtime relies on two communication channels for both servicing and handling specific RPC request. Both channels are setup as part of the sandbox startup process.

The native Java Runtime layer relies on C1 channel. The non-native layer makes use of C2 channel.

### 1.2.2 Virtual File System

GAE runtime implements a Virtual File System for arbitrary file access. For that purpose, GAE native layer intercepts all C library (`libc.so.1`) calls related to file / path, directory and file descriptor operations. This is accomplished with the use of a simple proxy mechanism (`LibcProxy` and `FDProxy` components). Actual libc call interception is implemented at the OS dynamic linker layer - the main `libajavaruntime.so` library simply redefines specific `libc.so.1` symbols:

```
01e30bb0 T open
01e30cd0 T open64
01e30df0 T access
01e30ef0 T read
01e31000 T write
```

GAE implementation for proxied library calls dispatches them over C1 communication channel to proper RPC services.

User application directory containing the unpacked WAR files becomes visible to GAE runtime upon mounting it at a predefined file system location (denoted by `/base/data/home/apps/app_name/app_version/` where `app_name` / `app_version` are application specific attributes). This mounting operation is conducted during the environment setup (new application deployment) and prior to servicing any user HTTP requests.

### 1.2.3 Socket proxy

GAE Java layer diverts all network related operations through RPC services. GAE installs a custom `SocketImplFactory` object for `java.net.Socket` and `java.net.ServerSocket`. A custom instance of `DatagramSocketImplFactory` is also installed for `java.net.DatagramSocket`. As a result, all socket related operations are proxied through a dedicated RPC service.

GAE also installs a custom URL handler for HTTP and HTTPS protocols. This is an instance of `com.google.apphosting.utils.security.urlfetch.URLFetchServiceStreamHandler` class. Its implementation diverts `java.net.URL` handling for HTTP and HTTPS protocols through GAE URLFetch service.

### 1.2.4 Class Sweeper

In GAE, all classes loaded by user applications are subject to the mandatory verification step (sweeping) performed prior to defining a given class in the JVM. The sweeping is conducted by a code of a `UserClassLoader` class and its `findClass` method in particular. It is done prior to the invocation of a native `defineClass` method of `java.lang.ClassLoader` class. This invocation chain is illustrated on Fig. 2.

`com.google.apphosting.runtime.security.UserClassLoader`

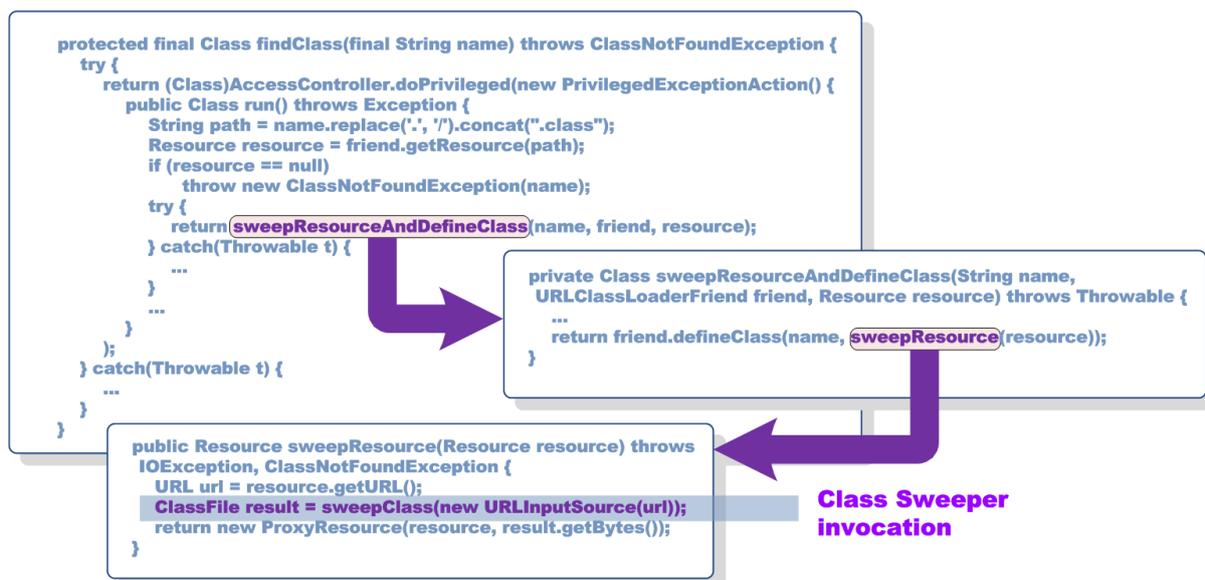


Fig. 2 The integration of a Class Sweeper into a class loading process.

Class sweeping forms a base security mechanism of the GAE Java sandbox. Implementation of several key GAE security concepts rely on it (i.e. Class Loaders, Reflection API, JRE Class White List).

During the sweeping process, the content of a Class file is inspected in order to both validate or enforce certain security restrictions defined by the GAE environment. The inspection process itself relies on a static analysis of Class files (its Constant Pool entries and Code attributes in particular).

The functionality of GAE Class sweeping is implemented by `com.google.apphosting.runtime.security.preverifier.PreVerifier` class. Its `sweep()` method has two arguments denoting arbitrary Java streams from where input Class data is read from and output Class bytes (the result of the sweeping) are written to:

```
public String sweep(InputStream input, OutputStream output)
```

Class sweeping can raise an exception if a given Class file does not meet certain security requirements of the GAE sandbox (i.e. invalid subclassing). If completed, the input Class bytes are either copied to the output stream without any modification or they are transformed according to the specific rules.

Class sweeping is implemented with the use of ASM [9], a Java bytecode manipulation and analysis framework. It is conducted with the help of several independent ASM modules described in a more detail below.

#### 1.2.4.1 Class Loader PreVerifier

The possibility to create arbitrary Class Loaders requires some modifications to their functionality in order to prevent against the malicious and privileged Class definitions in the JVM.

All Class Loader objects need to inherit from `java.lang.ClassLoader` class. As subclasses of the base system Class Loader class, user defined Class Loaders could directly invoke one of its protected `defineClass` methods that include a `ProtectionDomain` argument. For that reason, Class Loader classes are transformed in such a way, so that they extend a safe Class Loader class instead of the original superclass. A given safe Class Loader class either implements or transfers execution of certain security sensitive Class Loader methods such as `defineClass` to proper wrapper methods.

Table 1 presents JRE Class Loader classes and their corresponding safe GAE Class Loaders.

JRE Class Loader	Corresponding safe GAE Class Loader
<code>java.lang.ClassLoader</code>	<code>com.google.apphosting.runtime.security.shared.CustomClassLoader</code>
<code>java.net.URLClassLoader</code>	<code>com.google.apphosting.runtime.security.shared.CustomURLClassLoader</code>
<code>java.security.SecureClassLoader</code>	NOT SUPPORTED

Table 1 JRE Class Loader classes and their corresponding safe GAE Class Loaders.

Beside superclass replacement, Class Loader PreVerifier also inspects all instance method invocation instructions (*invokespecial* and *invokevirtual*) present in the code of user defined Class Loaders. Upon encountering the invocation of a `defineClass` method, its corresponding Java bytecode sequence is replaced in such a way, so that `safeDefineClass` method of a new superclass gets invoked. This mechanism is illustrated on Fig. 3.

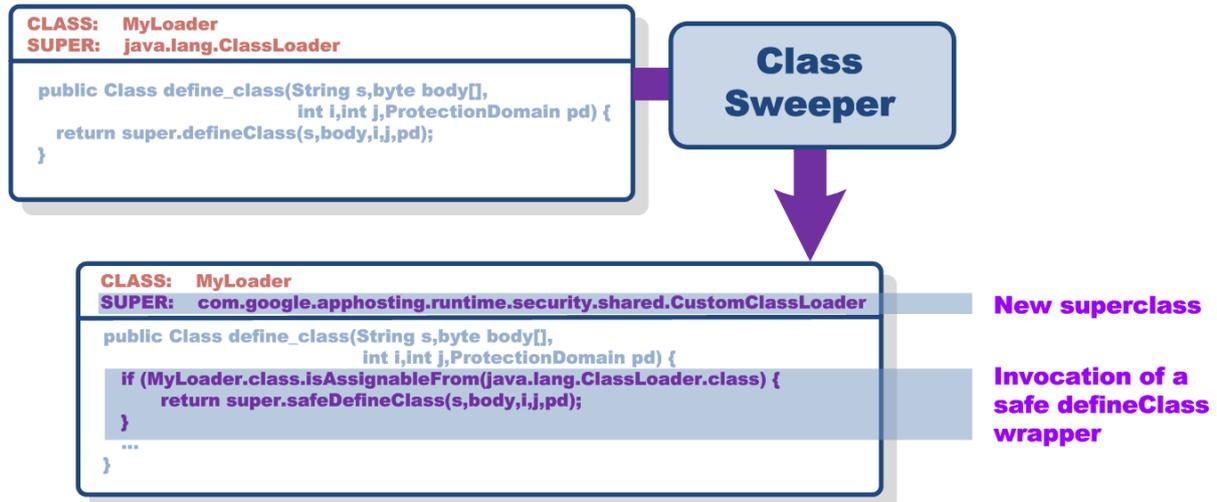


Fig. 3 Class Sweeper operation for custom, user defined Class Loaders.

The goal of `safeDefineClass` is to provide a safe replacement for `defineClass` method. Its implementation invokes Class Sweeper for all user defined classes. It also enforces a safe `ProtectionDomain` on them (user provided `ProtectionDomain` argument is ignored). Sample implementation of a safe replacement for a `defineClass` method used by `UserClassLoader` is illustrated on Fig. 4.

**com.google.apphosting.runtime.security.UserClassLoader**

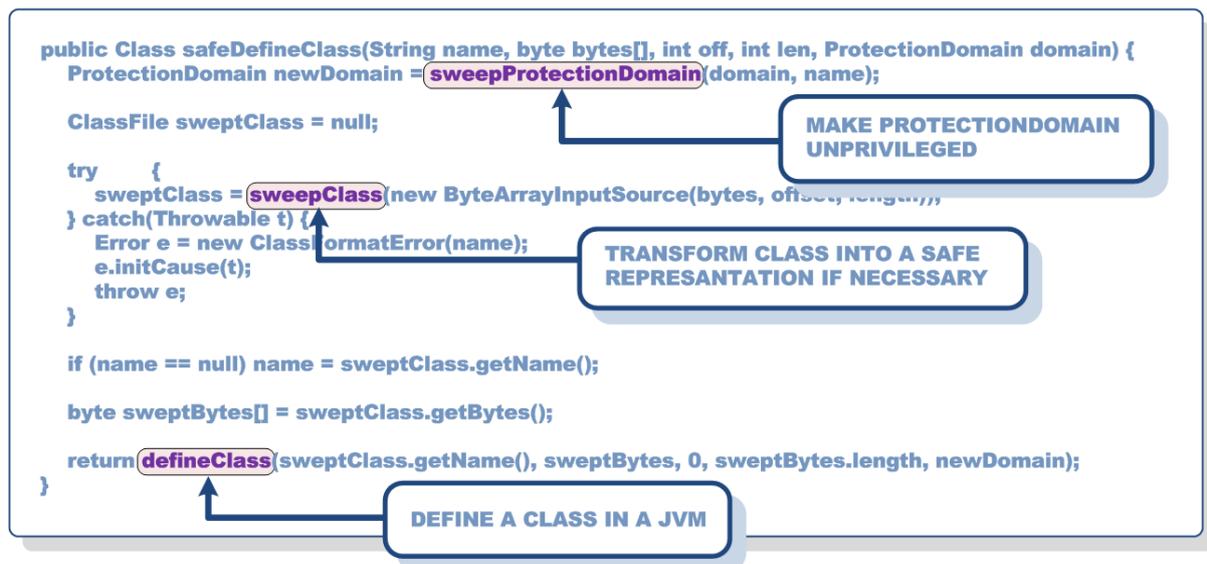


Fig. 4 The implementation of a safe replacement for a `defineClass` method used by `UserClassLoader`.

The `defineClass` method is not the only one that gets diverted to the new superclass implementation. Class Loader PreVerifier contains a list of guarded methods that are always dispatched from a given safe Class Loader superclass. The following methods were part of this list (among others):

- `PermissionCollection getPermissions(CodeSource codesource)`
- `Package definePackage(String s, String s1, String s2, String s3, String s4, String s5, String s6, URL url)`
- `Package getPackage(String s)`
- `Package[] getPackages()`
- `String findLibrary(String s)`

Class Loader PreVerfier also inspects `ldc` instructions in order to detect arbitrary loading of a method handles corresponding to either Class Loader's `defineClass` or one of its guarded methods. In Java 7, `ldc` instruction can push a reference to `java.lang.invoke.MethodHandle` object specified in a Constant Pool entry of a Class File [10] (the `CONSTANT_MethodHandle_info` structure). Class Loader Preverifier inspects `REF_invokeVirtual`, `REF_invokeStatic` and `REF_invokeSpecial` method handle kinds. Upon encountering a reference to a security sensitive method handle, its Constant Pool entry is replaced by a method handle corresponding to the safe superclass method. As a result, during runtime, `ldc` instruction will always push a safe replacement for a given Class Loader method (i.e. `safeDefineClass` instead of `defineClass`).

#### 1.2.4.2 Finalizer Visitor

Class Sweeper transforms the code of all user defined finalizers (`finalize()` methods) in such a way, so that they do nothing if invoked from within a finalizer handling system thread. The finalizer handling thread is detected by checking the name of the current thread. If it denotes "Finalizer" or "Secondary finalizer", the code is assumed to be executing in the context of a system finalizer thread.

#### 1.2.4.3 API Interjection and Interception

GAE implements a mechanism that allows for arbitrary modification or complete interception of JRE classes. API interjection mechanism makes it possible to invoke a given method prior to the invocation of another method. API interception allows to invoke a given method in place of another method.

API Interjection and Interception requires proper definition for interjected and intercepted classes and their methods (mirrors). In GAE, such definitions are maintained respectively under `interject` and `intercept` nodes of `com.google.apphosting.runtime.security.shared` package. In order to intercept or interject a method of a given class, one needs to define a class implementing this method in a `com.google.apphosting.runtime.security.shared.intercept` package. The package of the class needs to be changed to reflect that it is part of either the base `interject` and `intercept` package. The name of the class needs to have `_` character added to it. Finally, instance methods need to be changed to static ones and they also need

to have one extra argument added to the beginning of an arguments list. This is the original object for which the interjection / interception occurs (*this*).

The above rules are illustrated on Fig. 5. In order to intercept `getClassLoader()` method of `java.lang.Class` class, its mirror (`Class_ class`) needs to be defined in a `com.google.apphosting.runtime.security.shared.intercept.java.lang` package. It also needs to implement `public static ClassLoader getClassLoader(Class klass)` method.

### Original JRE class / method

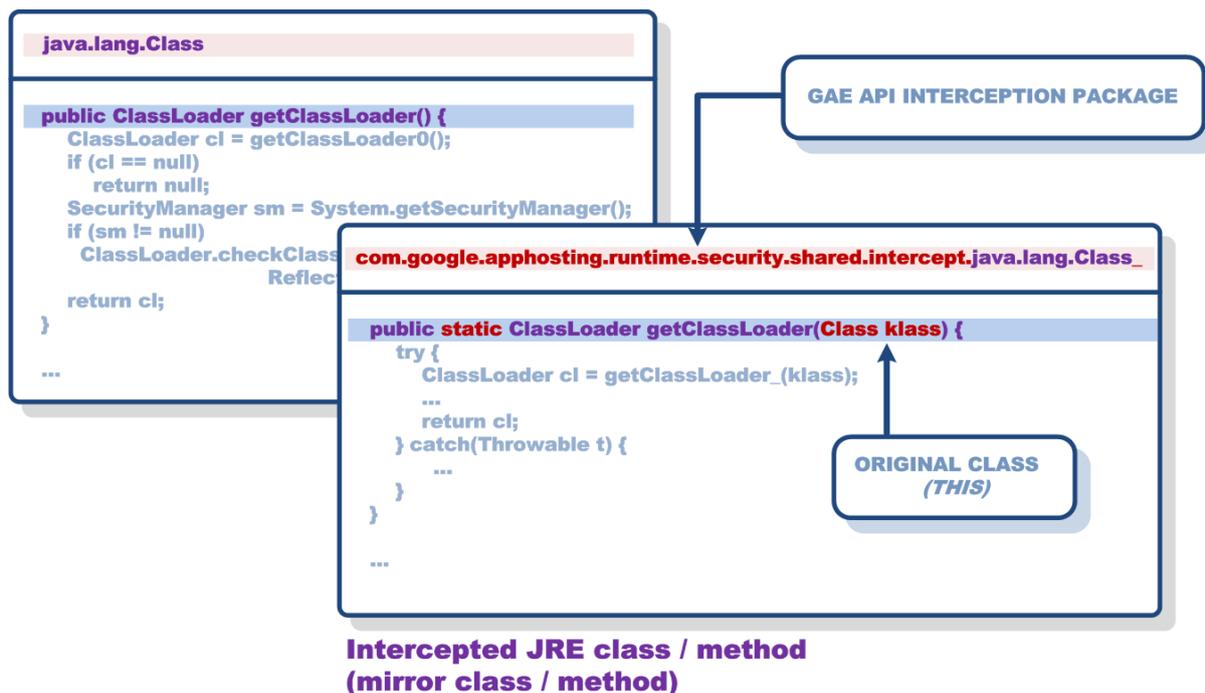


Fig. 5 Illustration of a JRE API Interception (`getClassLoader()` method of `java.lang.Class` class).

Interception API can be applied to methods, fields and constructors. Interjection API is supported for methods only.

GAE Class Sweeper applies proper transformation to the code of inspected classes in order to implement API interjection and interception functionality. If a given class referenced in user code has a mirror, every reference to it is replaced with a reference to that mirror. Similarly, instance invocations of methods (fields) from classes that have corresponding implementation (definition) in a mirror class are replaced with static invocations (access operation) of the mirrored method (field).

Reflection API and method handles are taken into account by the API Interception and Interjection mechanism. Thus, some base Reflection API classes (`java.lang.Class`, `java.lang.reflect.*`, `java.lang.reflect.invoke.*`) are subject to the interception themselves.

As of Oct 2014, GAE implemented API interception with respect to 160 JRE classes. API Interjection was conducted for 4 classes only. Table 2 presents information about selected classes and methods that were subject to the Interception API in GAE.

JRE Class	Method	Summary of a mirror class implementation
java.lang.Class	getClassLoader	For user classes, a reference to a defining Class Loader is returned. A NULL value is returned if a class comes from a PrivilegedClassLoader or a RuntimeClassLoader namespaces. In the latter case, the class needs to originate from a runtime-shared.jar location. In any other case, an AccessControlException is thrown.
	forName	Class.forName() is invoked in a doPrivileged method block with a loader argument corresponding to the Class Loader of a caller class.
	getProtectionDomain	A Protection Domain argument is returned for a given class.
	getMethod getMethods getDeclaredMethod getDeclaredMethod	For certain security sensitive java.lang.ClassLoader methods (i.e. defineClass, resolveClass), a corresponding safe method from a SafeClassDefiner class is returned. Otherwise, a corresponding JRE method is invoked in a doPrivileged method block.
java.lang.reflect.Method	getField getFields getDeclaredField getDeclaredFields getConstructor getConstructors getDeclaredConstructor getDeclaredConstructors	A corresponding JRE method is invoked in a doPrivileged method block
	Invoke	Any safe method gets invoked directly. Otherwise, a check is conducted whether a method is accessible <sup>10</sup> . If not, an exception gets thrown. If it is accessible and a corresponding method in a mirror class exists, a mirror method is invoked. Otherwise, the original method is invoked directly.

<sup>10</sup> an accessible Class member is either public, loaded or accessible by a user Class Loader.

<code>java.lang.reflect.Field</code>	<code>get</code>	A check is conducted whether a field is accessible. If not, an exception gets thrown. If it is accessible and a corresponding field in a mirror class exists, a value from a mirror field is read and returned. Otherwise, the value read from the original field is returned directly.
	<code>set</code>	A check is conducted whether a field is writable <sup>11</sup> . If not, an exception gets thrown. If it is writable, the original field is set with a given value.
<code>java.lang.Runtime</code>	<code>exit</code> <code>exec</code> <code>load</code> <code>loadLibrary</code> <code>addShutdownHook</code>	A <code>SecurityException</code> is thrown.
<code>java.lang.System</code>	<code>gc</code> <code>runFinalization</code> <code>runFinalizersOnExit</code>	Methods have an empty body (they do nothing).
	<code>setSecurityManager</code>	A <code>SecurityException</code> is thrown.

Table 2 Selected mirror classes and their methods.

It should be also noted that this was the API Interjection and Interception mechanism that lied at the core of a failure to reproduce the POCs illustrating the issues initially reported to Google<sup>12</sup>. We followed the same methodology for their development as in the case of SE-2013-01 project. This methodology assumed a minimum amount of work to be conducted during POCs development / testing in a target production cloud. Final tests were always scheduled to be conducted a few days prior to the actual reporting of the issues to the vendor<sup>13</sup>. As a result, most of our POCs were developed in a custom local GAE environment that was supposed to mimic Google's production environment as close as possible. In order to fulfill that requirement, we needed to know which classes were visible to the user code and what Class Loader namespaces they belonged to. This is why we invoked `Class.forName()` method on selected classes as well as `Class.getClassLoader()` call. We used the results obtained to setup the classpath of our local GAE environment. Unfortunately, we got fooled by GAE API Interception mechanism, which mislead us into thinking that:

- all classes are defined in a system (bootstrap) Class Loader namespace,
- all classes from a `RuntimeClassLoader` namespace are visible to user code.

The above conclusions were wrong. Intercepted `Class.getClassLoader()` call always returned NULL, instead of a real system GAE Class Loader reference. And by testing the visibility of random classes from a single JAR file of `RuntimeClassLoader` namespace,

<sup>11</sup> a writable Field cannot be final and it needs to be an accessible Class member.

<sup>12</sup> Issues 1-22 / unconfirmed Issues 23-35 reported to the company on 07-Dec-2014.

<sup>13</sup> this also applies to any purchase of a software license / service subscription.

certain peculiarities of `RuntimeClassLoader` implementation were missed (classes from a tested `runtime-shared.jar` location were allowed to load, but nothing else).

As a result, all POCs developed and tested in a local GAE environment worked fine locally, but many of them failed in a production environment<sup>14</sup>.

### **1.2.5 Class stubs**

For certain security sensitive classes, GAE provides dummy stub replacements. These are the classes that are defined under `com.google.apphosting.runtime.security.shared.stub` package node.

GAE stub classes contain dummy methods and initializers that don't do much beyond throwing an exception upon their invocation. A part of a stub class implementation corresponding to the `java.beans.Statement` class is provided below:

```
package com.google.apphosting.runtime.security.shared.stub.java.beans;

public class Statement {
    public Statement(Object obj, String s, Object aobj[]) {
        throw new NoClassDefFoundError("java.beans.Statement is a restricted class.
            Please see the Google App Engine developer's guide for more
            details.");
    }

    public Object getTarget() {
        throw new NoClassDefFoundError("java.beans.Statement is a restricted class.
            Please see the Google App Engine developer's guide for more
            details.");
    }
    ...
}
```

Regardless of the JRE security policy settings (`packages.access=sun.*, ...`), there are also stub classes defined for classes that have their origin in restricted packages such as `sun.misc.Unsafe`. Upon an attempt to load such a class, `UserClassLoader` will return a stub corresponding to the requested class instead of throwing a `SecurityException`.

### **1.2.6 Class Loader architecture**

GAE Java runtime creates several Class Loader namespaces, which provide natural isolation between user, runtime and system code.

User application code loading is always handled by an instance of `com.google.apphosting.runtime.security.UserClassLoader` class. This Class Loader is also set as current `Thread`'s context Class Loader. `UserClassLoader` namespace is always a subject to Class Sweeping. This forms a security boundary for GAE sandbox layer at a Class Loader level.

---

<sup>14</sup> 27 unexploitable issues with barely 7 candidates to work. We later proved that some of those "unexploitable" issues were still valid though (Issues 2, 5, 7, 12, 22, 23, 25, 26 and 27 in particular).

UserClassLoader also relies on two additional Class Loader's for class loading. An instance of `com.google.apphosting.runtime.security.RuntimeClassLoader` class is used to load GAE Java Runtime implementation classes. There is also an instance of `PrivilegedClassLoader` class, which is an internal class to `UserClassLoader` one. It handles classes implementing the API Interjection and Interception layer for Class Sweeper (all intercepted / interjected classes definitions). The actual implementation of the Class Sweeper engine and all Class Loaders described above is defined in JRE's `sun.misc.Launcher$AppClassLoader` namespace.

Class loading methods of both `UserClassLoader` and `RuntimeClassLoader` implement proper class filtering and checks that deny access to prohibited classes. By default, classes that are not part of the JRE or are on the Class Whitelist can be requested by user applications. User defined classes or classes defined by a user Class Loader are allowed to load.

Apart from that, the `RuntimeClassLoader` implements additional filtering that hides almost all of its classes' namespace from the `UserClassLoader`. The only exception are the classes that match the shared URLs codebase (`runtime-shared.jar`).

All GAE Class Loaders extend from `java.net.URLClassLoader` class. They have different privileges and codebases as illustrated on Fig. 6.

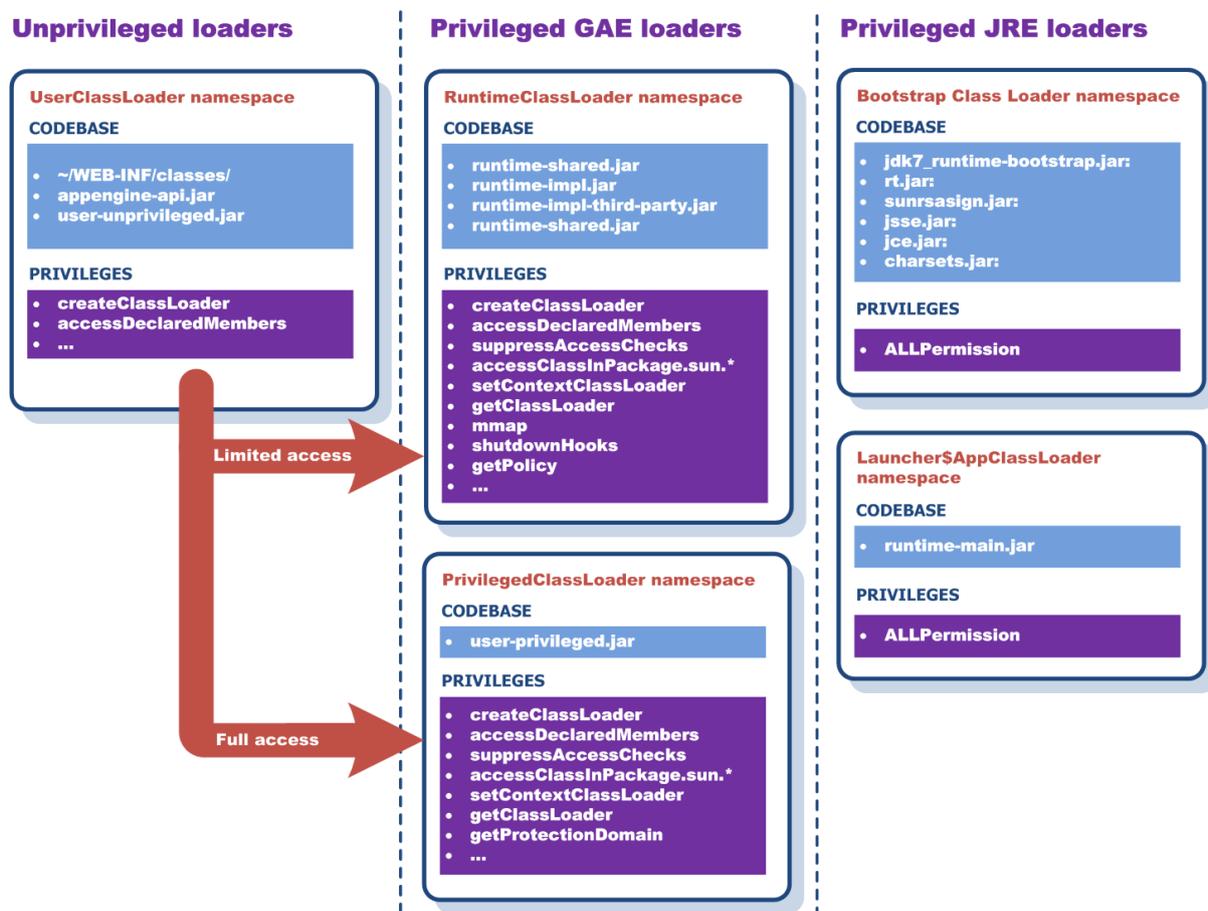


Fig. 6 GAE Class Loaders, their namespaces and privileges.

Finally, there is a bootstrap system Class Loader that primarily constitutes a namespace for JRE runtime classes and some GAE bootstrap classes as well. Classes defined in this namespace are fully privileged by default.

## 2 VULNERABILITIES

SE-2014-02 project resulted in a discovery of 31 security vulnerabilities in a Java security sandbox of Google App Engine service. Each of them is described in a detail in this part of the paper. Brief summary of all identified weaknesses can be found in APPENDIX A at the end of this report.

### 2.1 METHODOLOGY USED

As described in 1.1.5, GAE allows user code to create arbitrary Class Loader objects. GAE environment allows for a more privileged Reflection API access to JRE and application's classes as well (1.1.6). By default Java security sandbox neither grants Class Loader creation permission, nor the Reflection API access corresponding to the GAE model. This is due to the security risks they could pose to the sandbox.

Class Loader objects are quite powerful. They provide class definitions to the VM. They can specify permissions for loaded classes. Finally, they can also load native libraries into Java VM. These are just a few of the many reasons behind the requirement for the possession of a proper security privilege designating Class Loader creation in JRE. Class Loaders also provide the means to dynamically resolve unknown classes. With respect to this, their role in Java VM is similar to dynamic linkers' role in Unix.

Reflection API implementation allows for the violation of key Java security constraints such as data access protection and type safety. Insecure use of its functions conducted from within a privileged code can easily lead to the compromise of a Java security sandbox.

Vulnerable implementation of both Class Loaders and Reflection API has lead to many security vulnerabilities in the past [11][1].

Taking into account all of the above, both Class Loaders and Reflection API have been selected as the main focus of our research into GAE security (potential weak points).

### 2.2 DETAILS

#### 2.2.1 Issues 1, 2, 4 and 6

Class Sweeper tries to limit the security risk associated with a possibility to create arbitrary Class Loaders in a GAE environment. It does not however take into account the possibility to create an instance of a system class loader object with the use of Java Reflection API (core API and Java SE 7 based one<sup>15</sup>).

---

<sup>15</sup> the API implemented by the classes from `java.lang.invoke.*` package, referenced as a new Reflection API throughout this paper.

As a result, a straightforward combination of `getConstructor()` and `newInstance()` calls could be used to instantiate a `java.net.URLClassLoader` class (Issue 1). This is illustrated on Fig. 7.

#### java.net.URLClassLoader instantiation via getConstructor()/newInstance()

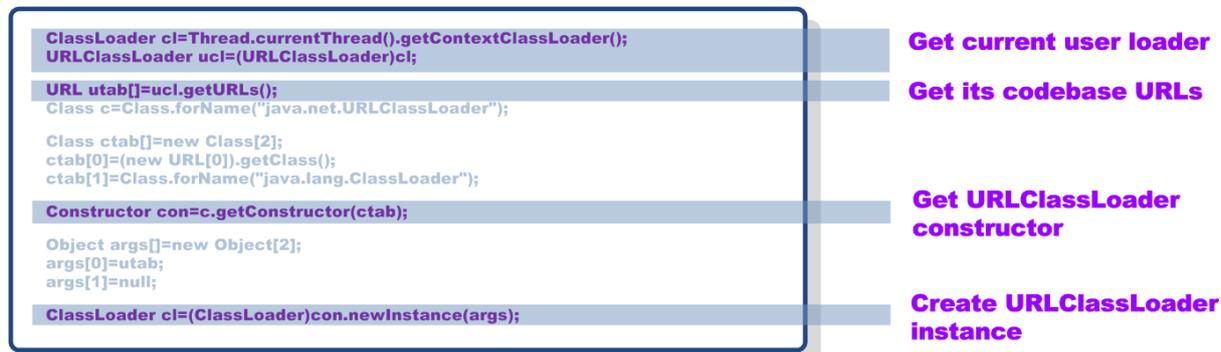


Fig. 7 An illustration of Issue 1

Similarly, a simple combination of `findConstructor()` and `invoke()` method calls of new Reflection API could be used to achieve exactly the same (Issue 6) as shown on Fig. 8.

#### java.net.URLClassLoader instantiation via findConstructor() MethodHandle

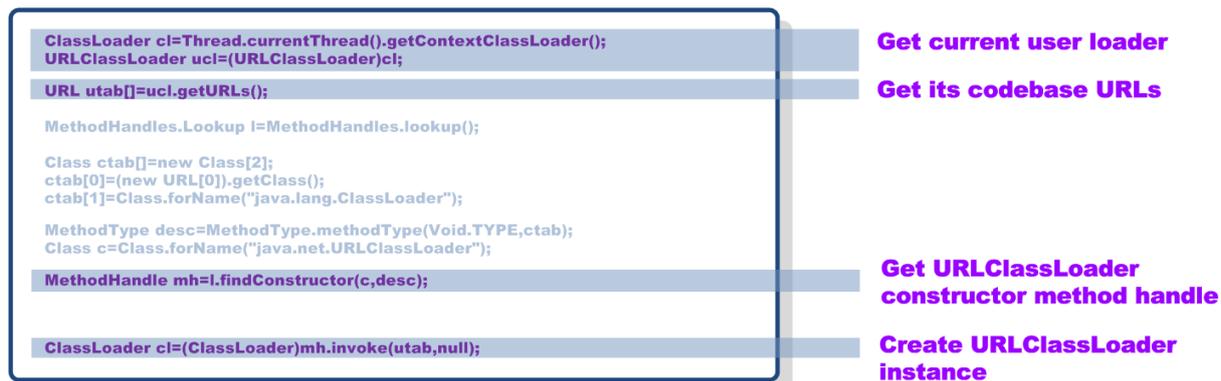


Fig. 8 An illustration of Issue 6

Additionally, GAE does not take into account the possibility to create an instance of a system class loader object with the use of a Reflection API invocation embedded in JRE code. This lies at the core of both Issues 2 and 4. Issue 2 exploits the implementation of a whitelisted `java.security.Provider.Service` class for a system Class Loader instantiation (Fig. 9).

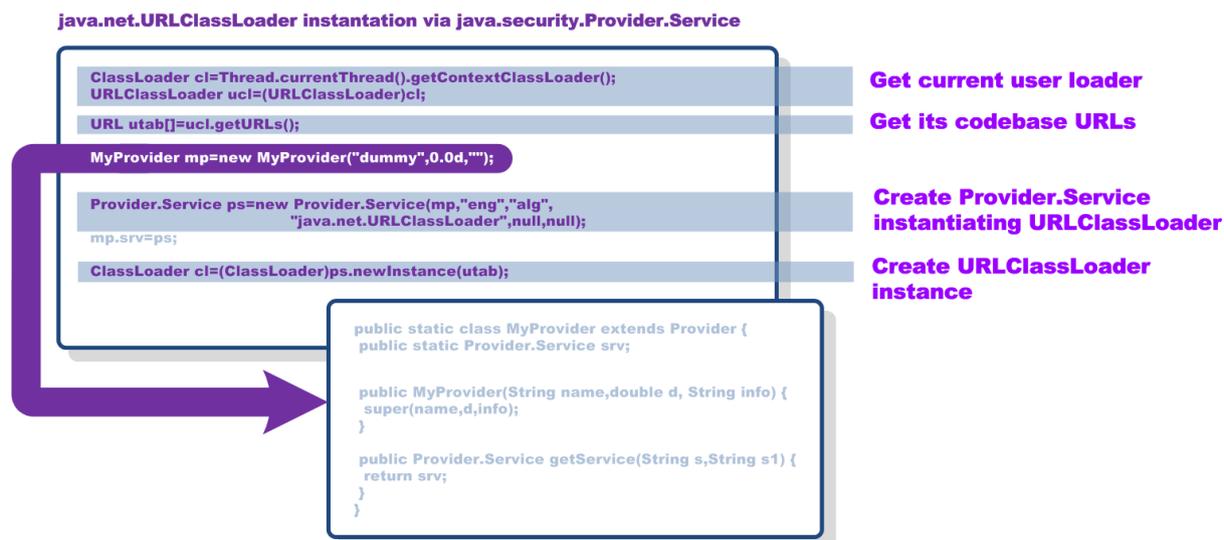


Fig. 9 An illustration of Issue 2.

Issue 4 allows to create an instance of a system class loader object with the use of a `java.beans.XMLDecoder` class. A specially crafted XML file provided as an input to `java.beans.XMLDecoder` object instance can result in an execution of arbitrary Java methods and constructors outside of a GAE control.

Issue 2 demonstrates an attack technique against an arbitrary Java class whitelisting mechanism as a vulnerability that was used to bypass Java API whitelisting rules of Oracle Java Cloud Software<sup>16</sup> [12]. Issue 4 is exactly the same vulnerability as the one used in Oracle Java Cloud environment<sup>17</sup>. Its more detailed description can be found in our Java SE security research report from 2012 [13].

It's worth to note that `URLClassLoader` objects instantiated with the use of either Issue 1, 2, 4 or 6 are fully functional. Arbitrary user provided code (classes) could be loaded and executed through them. The most important thing is however related to the fact that `URLClassLoader` namespace is not a subject to any code transformation (sweeping). Thus, a sole instance of an `URLClassLoader` object under attacker's control constitutes a successful escape of a GAE Java security sandbox<sup>18</sup> imposed by the Class Sweeper and associated API Interjection and Interception mechanism in particular (escape of `UserClassLoader` namespace).

### 2.2.2 Issue 3

A replacement of a `forName` method of `java.lang.Class` class implemented by the GAE API Interception mechanism contains an insecure invocation of an actual `Class.forName()` Reflection API call. This is illustrated on Fig. 10.

<sup>16</sup> Static checks imposed by a Java API whitelisting rules of Oracle Java Cloud Service could be also bypassed through a Reflection API trampoline in a system code (Issue 18).

<sup>17</sup> Issue 30 of SE-2013-01 project (Java API whitelisting rules bypass through XMLDecoder).

<sup>18</sup> This should not be confused with a JRE sandbox as it still needs to be escaped (i.e. Security Manager needs to be turned off).

`com.google.apphosting.runtime.security.shared.intercept.java.lang.Class_`

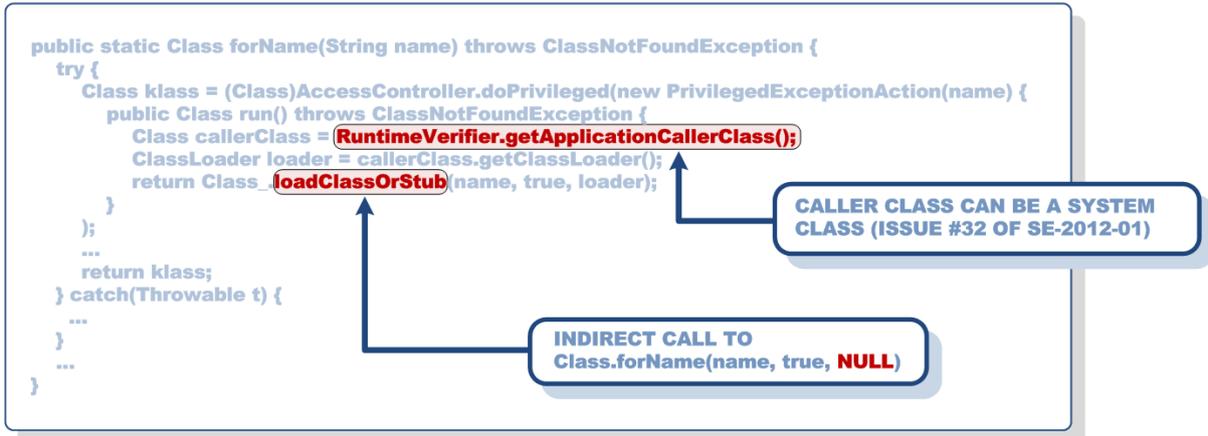


Fig. 10 An illustration of Issue 3.

A static, no-argument `forName()` method of `com.google.apphosting.runtime.security.shared.intercept.java.lang.Class_` class contains a `Class.forName()` method invocation done from within a `doPrivileged` method block. Its 3rd argument denotes a Class Loader of a caller's class obtained with the use of a `RuntimeVerifier.getApplicationCallerClass()` call. It can be easily enforced to denote a system (NULL) CL namespace with the use of a new Reflection API and `invokeWithArguments` method in particular. As a result, arbitrary access to restricted classes could be gained as the privileges of the `Class_` class allow for access to classes from a `sun.*` package (`PrivilegedClassLoader` namespace).

It's worth to note that Issue 3 demonstrated exactly the same attack against security sensitive Reflection API calls as Issue 32 of SE-2012-01 project. Oracle's fix for Issue 32 relies on a binding of the `MethodHandle` object to the caller of a target method / constructor if it denotes a potentially dangerous Reflection API call. This binding has a form of injecting extra stack frame from a caller's Class Loader namespace into the call stack prior to issuing a security sensitive method call [14]. GAE interception API broke Oracle's fix. Although, a security of the intercepted `Class.forName()` call still relied on a caller class, no `MethodHandle` binding was performed. As a result, a security vulnerability was introduced.

### 2.2.3 Issue 5

Similarly to Issue 3, the implementation of an `invoke` method of `com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.Method_` class contains an insecure invocation of `Method.invoke()` Reflection API call. It constitutes the replacement for a real `Method.invoke()` Reflection API call defined in a `PrivilegedClassLoader` namespace and is part of the GAE API interception mechanism.

The problem with an arbitrary method invocation conducted from within a more privileged class is that it can be abused by an attacker to achieve a complete GAE java security escape as described in 3.1.3.1.

## 2.2.4 Issue 7

As indicated in 1.2.4.1, GAE intercepts certain `java.lang.ClassLoader` method invocations in order to protect against arbitrary privileged class definitions in the JVM. A given safe Class Loader class either implements or transfers execution of certain security sensitive Class Loader methods such as `defineClass` to proper wrapper methods. Unfortunately, GAE does not take into account the possibility to obtain a reference to a protected Class Loader method with the use of a `findSpecial()` method call of a new Reflection API. As a result, a valid method handle to security sensitive `defineClass` method could be obtained and called. This condition can be exploited to achieve a complete GAE security sandbox escape (arbitrary class definition in a privileged protection domain with no Class Sweeper in place).

## 2.2.5 Issues 8 and 10

GAE Interception API implementation for `getMethod()` and `getDeclaredMethod()` calls of `java.lang.Class` class does not return references to certain security sensitive methods of Class Loader classes such as `defineClass`, `resolveClass` and `findLoadedClass`. Instead, a reference to a safe replacement method from `com.google.apphosting.runtime.security.shared.SafeClassDefiner` class is provided (`safeDefineClass`, `safeResolveClass` and `safeFindLoadedClass` respectively). This is illustrated on Fig. 11.

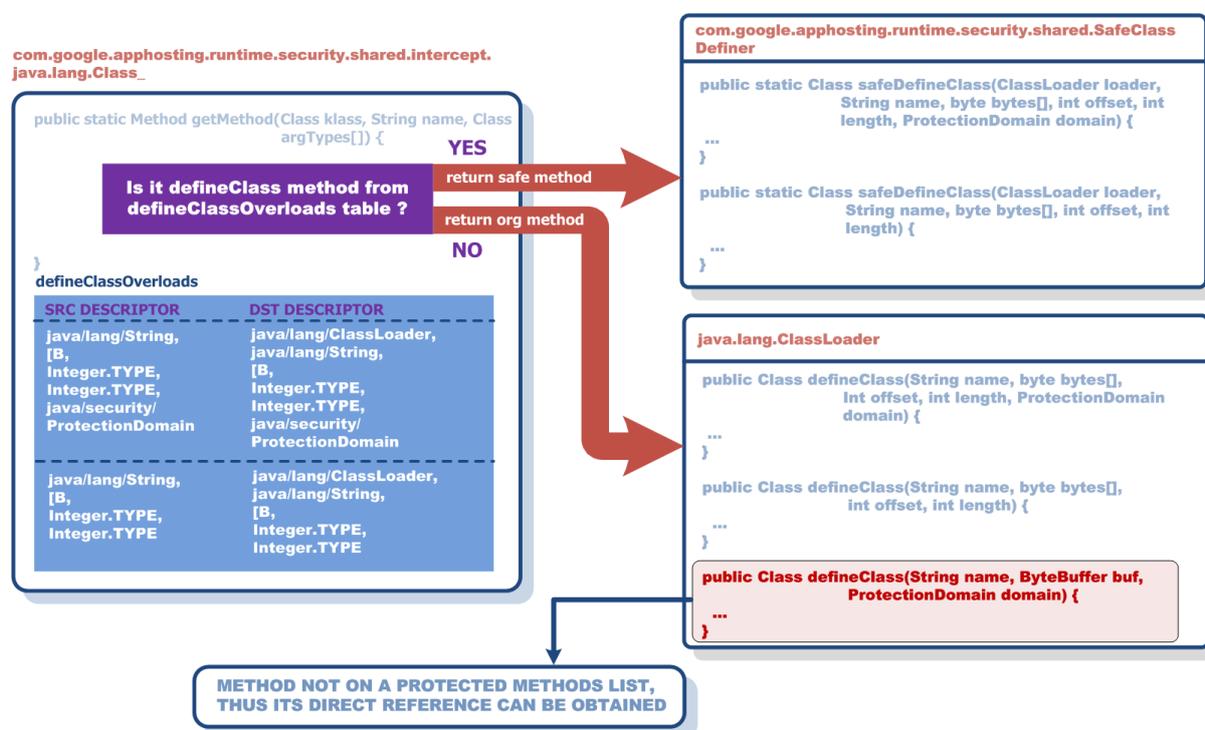


Fig. 11 An illustration of Issue 8.

The `Class_` interception class makes use of the internal `defineClassOverloads` map to keep track of `defineClass` methods that are a subject to the abovementioned

replacement. While this map contains information about two most frequent instances of a `defineClass` method, it misses information about the following entry:

```
protected final Class defineClass(String s, ByteBuffer bytebuffer,
ProtectionDomain protectiondomain)
```

As a result, it is possible to invoke `getDeclaredMethod` of `java.lang.Class` class and obtain a direct reference to a security sensitive `defineClass` method of `java.lang.ClassLoader` class (Issue 8).

Finally, GAE Interception API implementation for `getDeclaredMethods()` call of `java.lang.Class` class neither filters, nor replaces any of the methods in a returned array. This can be exploited to obtain a direct reference to a security sensitive `defineClass` method of `java.lang.ClassLoader` class (Issue 10).

Due to the limits imposed by GAE Interception API and the `invoke` method of `java.lang.reflect.Method` class in particular, method references obtained with either Issue 8 or 10 cannot be directly called from within `UserClassLoader` namespace. They need to be accompanied by additional vulnerabilities in order to make use of the obtained `defineClass` methods.

### **2.2.6 Issues 9, 11, 15 and 16**

GAE Interception API implementation for method handle related operations does not intercept all of the `java.lang.invoke.MethodHandles.Lookup` class methods. This in particular includes methods that allow for a transformation of core Reflection API objects (instances of `java.lang.reflect.Method` / `java.lang.reflect.Field` class) into method handles.

The implementation of the following methods is missing from the `com.google.apphosting.runtime.security.shared.intercept.java.lang.invoke.MethodHandles.Lookup` class:

- `unreflect` (Issue 9)
- `unreflectSpecial` (Issue 11)
- `unreflectGetter` (Issue 15)
- `unreflectSetter` (Issue 16)

GAE follows JRE approach to method handles security. It assumes that a method handle obtained with the use of any of the intercepted `MethodHandles.Lookup` class methods (i.e. `findVirtual`, `findStatic`) is already safe for use. Thus, no security checks are implemented during method handle invocations. In JRE, proper security checks are also conducted at the time of a given method handle creation, but not at the time of its invocation. Thus, a valid method handle reference is always invocable in GAE.

Due to the incomplete interception of the methods of `java.lang.invoke.MethodHandles.Lookup` class, GAE restrictions imposed on Reflection API objects can be successfully bypassed. These restrictions limit the possibility to

invoke methods and access fields that are not public and are part of a GAE / JRE code (not defined by user application code or any user Class Loaders created by it). This also includes fields and methods with an overridden access<sup>19</sup>.

By turning method and field references into method handles, one can successfully escape GAE security sandbox. As a result, security sensitive methods such as `defineClass` method of `java.lang.ClassLoader` class obtained with either Issue 8 or 10 can be invoked. All that is needed for that purpose is a proper transformation of a given method into a corresponding method handle as illustrated on Fig. 12.

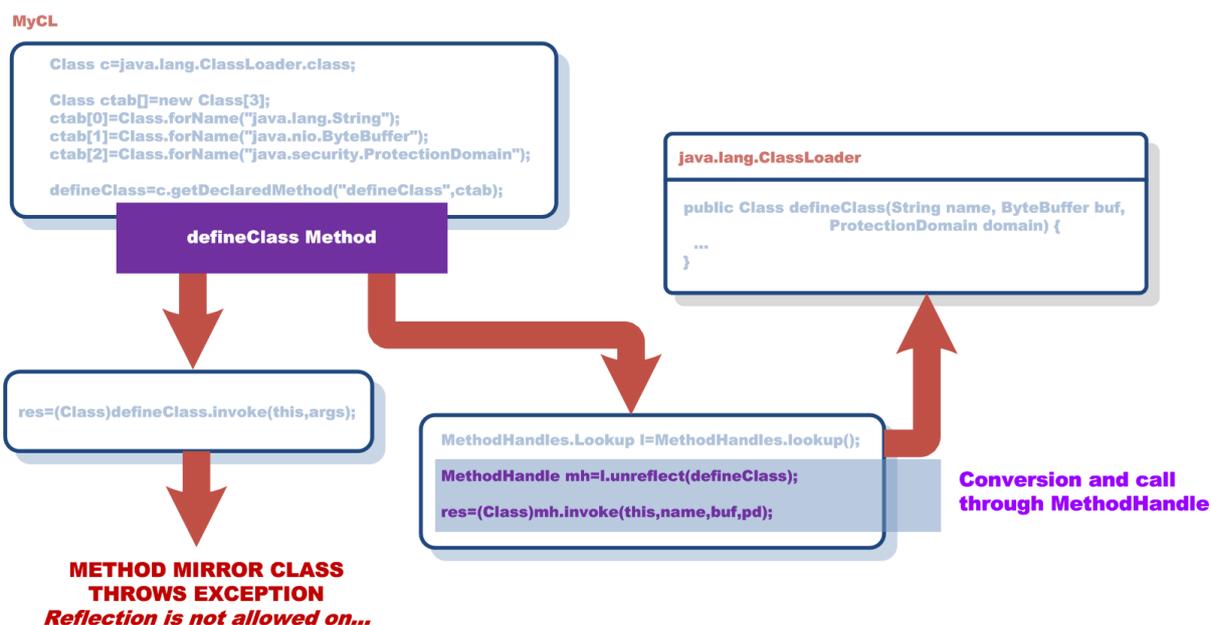


Fig. 12 An illustration of Issue 9.

Similarly, a restricted `java.lang.reflect.Field` object can be read or set upon turning it into a method handle. This can be in particular helpful if a security sensitive field (i.e. `unsafe` field of `java.util.Random`) is to be read from a system class.

It should be mentioned that Issue 9 is similar to Issue 64 we reported to IBM in May 2013 [15].

### 2.2.7 Issue 12 and 14

As described in 1.2.4.1, Class Loader PreVerifier inspects `ldc` instructions in order to detect arbitrary loading of a method handles corresponding to either Class Loader's `defineClass` or one of its guarded methods. It changes any potentially unsafe method handle into a corresponding safe replacement.

Similarly, GAE Interception API implementation for method handle lookup operations returns safe replacements for certain security sensitive method handles such as those corresponding to the `defineClass` method of `java.lang.ClassLoader` class.

<sup>19</sup> Such as those for which `setAccessible()` method of `java.lang.reflect.AccessibleObject` was invoked.

The intercepted `findVirtual()` method first checks whether a given method lookup operation is done with respect to a Class Loader object. If this is the case, `lookupSafeDefineClass` method is invoked, which tries to find a safe replacement for a looked up method handle, but only if it corresponds to a `defineClass` method. This is illustrated on Fig. 13.

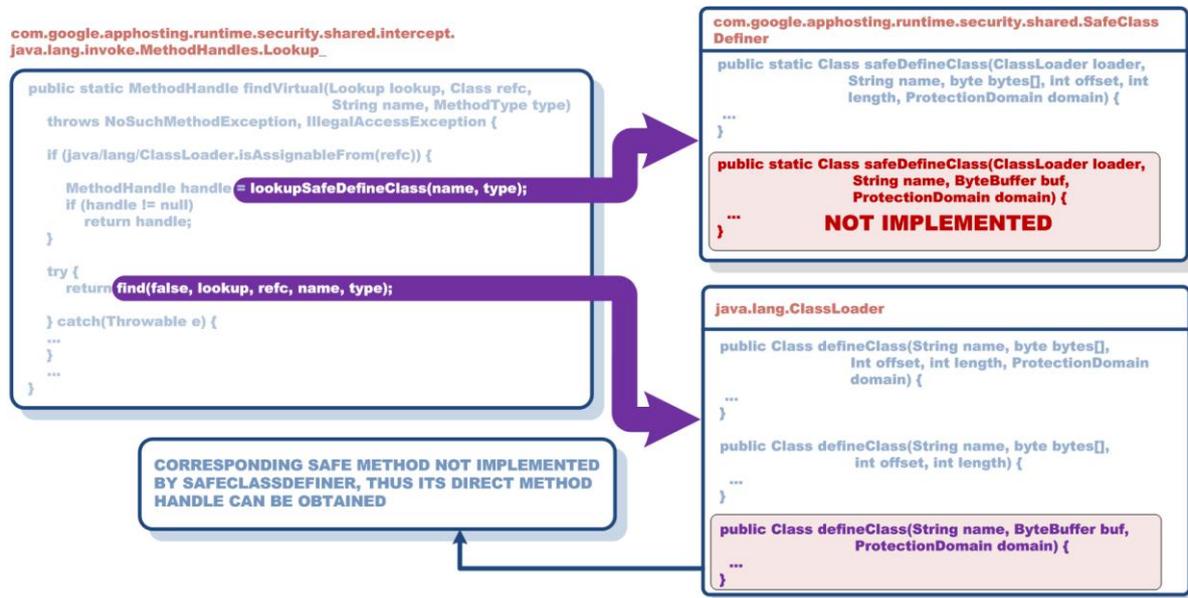


Fig. 13 An illustration of Issue 12.

The search for a safe replacement method handle is done through an internal lookup object and with the use of a `findVirtual()` call as well. The lookup is however always done against `com.google.apphosting.runtime.security.shared.SafeClassDefiner` class and for a `safeDefineClass` method name. Its type descriptor has one extra argument added to the beginning of an arguments list, so that it meets the requirement for the intercepted / interjected method.

Unfortunately, `SafeClassDefiner` class does not implement<sup>20</sup> all security relevant methods corresponding to the `defineClass` of `java.lang.ClassLoader` class. This in particular considers the following methods:

```
public static Class safeDefineClass(ClassLoader, String, ByteBuffer, ProtectionDomain)
public static Class safeDefineClass(ClassLoader, byte[], int, int)
```

The two missing `safeDefineClass` methods lie at the core of Issues 12 and 14. If a search for a safe replacement method handle cannot find it in the `SafeClassDefiner` class, the method handle lookup operation does the search against the original class. As a result, actual `defineClass` method handle is returned by to the caller. This method handle can be further invoked without any restrictions as explained in 2.2.6. In case of Issue 12, this can immediately lead to the complete escape of a GAE security sandbox (`defineClass` invocation with an arbitrary, user provided `ProtectionDomain` argument).

<sup>20</sup> no implementation of a given method replacement by `SafeDefineClass` implicates no interception of the original method.

### **2.2.8 Issue 13**

The ability to define custom Class Loader objects in GAE implicates the ability to call protected methods of `java.lang.ClassLoader` class. For that reason, Class Loader PreVerifier changes the code of user defined Class Loader's so that certain security sensitive method invocations are always dispatched from a given safe Class Loader superclass.

Class Loader PreVerifier does not however take into account a `findSystemClass` method of `java.lang.ClassLoader` class. As a result, arbitrary system classes can be loaded by user code, including classes not present on JRE Class Whitelist.

### **2.2.9 Issues 17 an 18**

The implementation of a GAE Interception mechanism for Reflection API calls allow for arbitrary access to members of system classes denoted as prohibited by a Java security policy<sup>21</sup> (Issue 17). This both violates and weakens a Java security sandbox.

The following methods of `java.lang.Class` class were found to be affected:

- `getField`
- `getFields`
- `getDeclaredField`
- `getDeclaredFields`
- `getMethod`
- `getMethods`
- `getDeclaredMethod`
- `getDeclaredMethods`
- `getConstructor`
- `getConstructors`
- `getDeclaredConstructor`
- `getDeclaredConstructors`

Similarly, the implementation of a GAE Interception mechanism for Reflection API calls allow for arbitrary access to declared members of system classes (Issue 18). This also violates and weakens a Java security sandbox.

The following methods of `java.lang.Class` class were found to be affected:

- `getDeclaredField`
- `getDeclaredFields`
- `getDeclaredMethod`
- `getDeclaredMethods`
- `getDeclaredConstructor`
- `getDeclaredConstructors`

Regardless of the fact that there are multiple, separate method calls affected by each of the issues, and due to the fact that original JRE calls corresponding to the affected methods

---

<sup>21</sup> in GAE, `package.access=sun. ,\` etc.

contain proper security checks (missing in a code of the intercepted methods), we decided to treat Issues 17 and 18 as single ones instead of assigning a separate issue to each of the affected calls. The reason for it was the use of a security check in a form of an `isInspectable` method call of `com.google.apphosting.runtime.security.shared.RuntimeVerifier` class in each of the affected methods. We came to the conclusion that this was likely an implementation of this method that lacked proper `checkMemberAccess` and `checkPackageAccess` calls, not the implementation of the affected Reflection API methods.

### 2.2.10 Issue 19

GAE Interception API implementation for a `getProtectionDomain()` method of `java.lang.Class` class always returns an instance of a `ProtectionDomain` object for a given class argument. This is due to the invocation of a `getProtectionDomain()` call itself done from within a `doPrivileged` method block as illustrated on Fig. 14.

**`com.google.apphosting.runtime.security.shared.intercept.java.lang.Class_`**

```

public static ProtectionDomain getProtectionDomain(Class klass) {
    try {
        ProtectionDomain pd = (ProtectionDomain)AccessController.doPrivileged(new PrivilegedAction(klass) {
            public ProtectionDomain run() {
                return klass.getProtectionDomain();
            }
        });
    } catch (Throwable t) {
        ...
    }
    return pd;
}

```



Fig. 14 An illustration of Issue 19.

This is also in contrary to a JRE implementation of the `getProtectionDomain()` call. The latter always checks for a proper permission prior to giving access to the security sensitive `ProtectionDomain` object as it can both leak information about security boundaries of a given code (its permissions, classpath JAR files and their locations), but also facilitate certain privilege elevation attacks<sup>22</sup>.

### 2.2.11 Issue 20

Issue 19 revealed information about file permissions granted to user application classes:

```

class MyFirstJAppServlet
ProtectionDomain
(file:/base/data/home/apps/s~myfirstjapp/1.379850528770929561/WEB-INF/classes/ <no
signer certificates>)
com.google.apphosting.runtime.security.UserClassLoader@877c09
<no principals>
java.security.Permissions@b06677 (

```

<sup>22</sup> those attacks that rely on a modification of the `permissions` field of a `ProtectionDomain` object associated with a user class.

```
("java.lang.RuntimePermission" "createClassLoader")
("java.lang.RuntimePermission" "accessDeclaredMembers")
...
("java.io.FilePermission" "/base/java7_runtime/prebundled/user-unprivileged.jar"
"read")
("java.io.FilePermission" "/base/jre7/lib/rt.jar" "read")
("java.io.FilePermission" "/base/java7_runtime/runtime-shared.jar" "read")
("java.io.FilePermission" "/base/java7_runtime/prebundled-connector-j/jdbc-mysql-
connector.jar" "read")
...
)
```

It turned out that user applications can obtain runtime classes used in a GAE environment as well as some code implementing a GAE sandbox itself. This information leak makes it possible for an attacker to both verify the patching status of the JRE as well as help identify some security vulnerabilities in a GAE code such as Issue 24.

### **2.2.12 Issue 21**

Exploitation of Issue 20 revealed an additional vulnerability. It turned out that a JRE runtime class base used in a GAE environment is 1+ years old.

In Sep 2013, Oracle changed the implementation of new Reflection API (JDK 7 Update 40). The contents of `java.lang.invoke` package from `/base/jre7/lib/rt.jar` file indicates that the JRE used in GAE is prior to that time (no `LambdaForm` class, missing security checks in `MethodHandles.Lookup` class implementation, etc.).

The above means that the environment was potentially vulnerable to 100+ unpatched security vulnerabilities [16]. This also means that Issue 69 published in Oct 2013 [17] should also work in a GAE environment upon some modification<sup>23</sup>.

### **2.2.13 Issues 22, 23, 25, 26 and 27**

Any vulnerability that allows for an arbitrary escape of a GAE Class Loader restrictions makes it possible to gain access to the classes from a `RuntimeClassLoader` namespace.

This is in particular relevant as in GAE, a reference to a given Class Loader instance can be obtained by the means of a `getClassLoader()` method. This method needs to be invoked in an escape Class Loader namespace and on a `ProtectionDomain` object associated with a class defined by a loader of which reference is to be obtained. Thus, a reference to the `RuntimeClassLoader` instance can be obtained through any of the visible `runtime-shared.jar` classes. It can be further used to load arbitrary GAE runtime classes, including those not visible to the `UserClassLoader` namespace.

Access to the `RuntimeClassLoader` classes allows to exploit security vulnerabilities present in their code. These classes are defined in a more privileged `Protection Domain` than user applications classes (Fig. 6). Thus, any security issue affecting the `RuntimeClassLoader` namespace creates a potential for a privilege elevation attack.

---

<sup>23</sup> we verified that this was actually the case in Mar 2015.

The `org.mozilla.javascript.tools.shell.JavaPolicySecurity` class is a good example for that. This class embeds an insecure implementation of an internal Class Loader instance that makes use of a user provided Protection Domain argument in its `defineClass` method (Issue 22). This is illustrated on Fig. 15.

### **org.mozilla.javascript.tools.shell.JavaPolicySecurity\$Loader**

```
private static class Loader extends ClassLoader
implements GeneratedClassLoader {

    private ProtectionDomain domain;

    public Class defineClass(String name, byte data[]) {
        return super.defineClass(name, data, 0, data.length, domain);
    }

    ...

    Loader(ClassLoader parent, ProtectionDomain domain) {
        super(parent == null ? getSystemClassLoader() : parent);
        this.domain = domain;
    }
}
```



Fig. 15 An illustration of Issue 22.

Additionally, there are several classes that implement certain Reflection API calls in an unsafe way. This in particular includes arbitrary invocation of the `invoke` method of `java.lang.reflect.Method` class. Such an invocation is used by the following `RuntimeClassLoader` classes:

- `com.google.common.reflect.Invokable$MethodInvokable` (Issue 23)
- `org.apache.commons.beanutils.MethodUtils` (Issues 25 and 26)
- `org.codehaus.jackson.map.introspect.AnnotatedMethod` (Issue 27)

### **2.2.14 Issue 24**

Issue 24 is a vulnerability which can be exploited to both gain access to a `RuntimeClassLoader` namespace and to implement a complete GAE security sandbox escape. It stems from an insecure use of `invoke` method of `java.lang.reflect.Mehod` class in `com.google.apphosting.util.UserClassLoaderHelper` class.

The interesting thing about `UserClassLoaderHelper` class is that it is defined by a bootstrap Class Loader and is not available to user code by default. The vulnerable code can be however reached through a `javax.el.BeanELResolver` class, which belongs to a user visible part of a `RuntimeClassLoader` namespace. This is illustrated on Fig. 16.

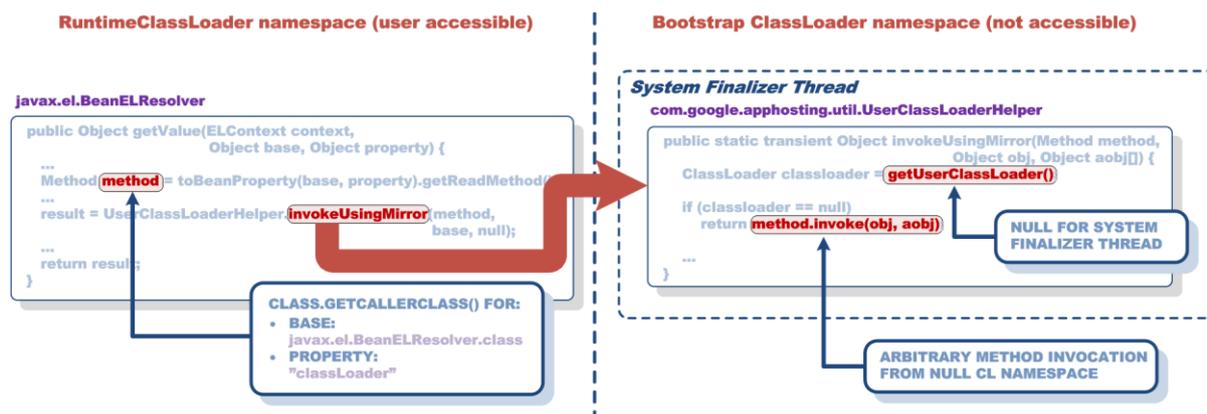


Fig. 16 An illustration of Issue 24.

The code of `BeanELResolver` class calls `invokeUsingMirror` method of a `UserClassLoaderHelper` class. This invocation is however limited to Beans' setter and getter methods only. What this means is that the called methods need to either:

- have a name starting with a *get* string and be a no-argument methods,
- have a name starting with a *set* string and be a one argument methods.

Additionally, the `invokeUsingMirror` method will directly invoke a target method only if current Thread's context Class Loader value is NULL. In any other case, it will be proxied through the intercepted Reflection API class (will be a subject to GAE restrictions).

The above prerequisites are not a big obstacle though. The vulnerable `invoke` method of `com.google.apphosting.util.UserClassLoaderHelper` class can be successfully exploited in two steps. Both steps need to be conducted from within a finalizer of an arbitrary object. The reason for it is the requirement for a NULL value of current Thread's context Class Loader. In JRE, this value is always NULL when a system finalizer thread processes (invokes) arbitrary finalizers (`finalize()` methods).

In step 1, `BeanELResolver` class can be used to invoke a getter method for a `classLoader` property of `BeanELResolver` class itself. As a result, `getClassLoader()` method of `java.lang.Class` will be invoked for `BeanELResolver` class. The result of this call will be the value of a `RuntimeClassLoader`.

In step 2, access to `RuntimeClassLoader` namespace can be used for a direct invocation of `invokeUsingMirror` method of `com.google.apphosting.util.UserClassLoaderHelper` class. As a result, an arbitrary method invocation could be achieved from within a privileged Class Loader namespace. This condition can be further exploited to achieve a complete GAE security sandbox escape with the use of a technique presented in paragraph 3.1.3.2.

### 2.2.15 Issues 28 and 29

As described in 1.2.4.2, GAE changes the code of `finalize()` methods in order to protect against arbitrary execution of user provided finalizers.

The above restriction can be bypassed with the use of a whitelisted `java.io.zip.ZipFile` system class that can be exploited to invoke a user provided code as part of its `finalize()` method implementation (Issue 28). In a `ZipFile` case, its `close()` method is called inside a finalizer. Thus, all that is needed to call arbitrary user provided code inside a system finalizer is to extend a `ZipFile` class and provide arbitrary implementation for its `close()` method. This is illustrated on Fig. 17.

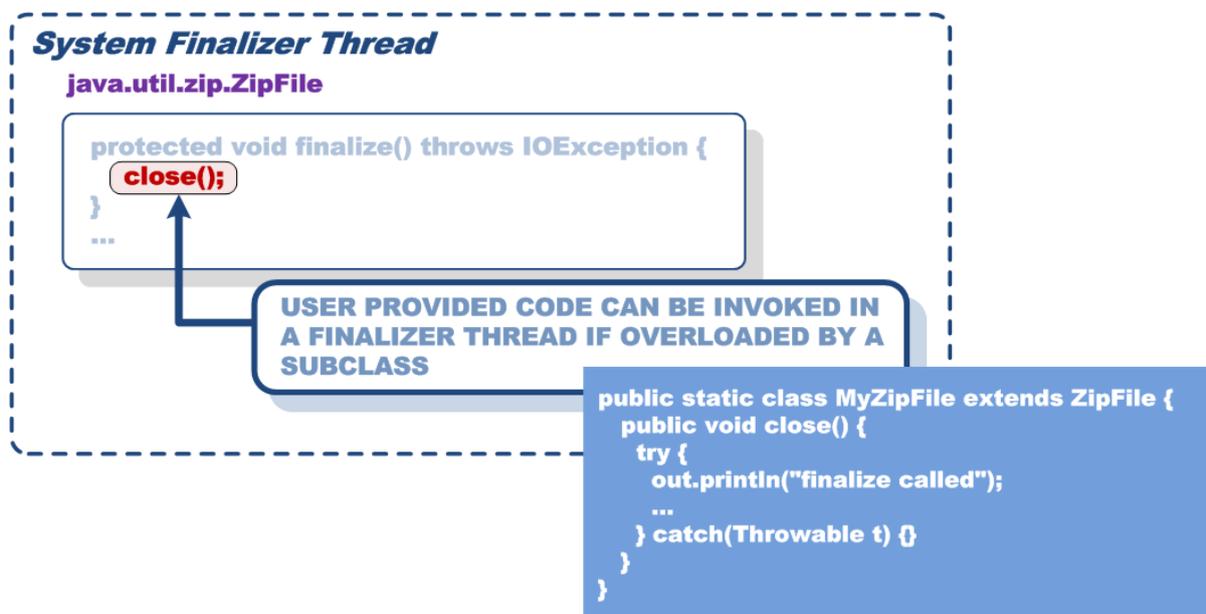


Fig. 17 An illustration of Issue 28.

Paragraph 1.1.4 also indicates that Java SE API methods implementing implicit Garbage Collection (GC) calls are intercepted in a way that makes it difficult for user code to control the GC process. This can be bypassed with the use of a system `java.nio.Bits` class (Issue 29). The code containing arbitrary invocation of a `System.gc()` call can be triggered with the use of the following code sequence:

```
ByteBuffer.allocateDirect(0x10000000);
```

The `allocateDirect` method of `java.nio.ByteBuffer` class allocates an instance of a `DirectByteBuffer` class. Its constructor invokes `reserveMemory` method of `java.nio.Bits` class for that purpose:

```
static void reserveMemory(long size, int cap) {
    ...
    if (cap <= maxMemory - totalCapacity) {
        reservedMemory += size;
        totalCapacity += cap;
        count++;
        return;
    }

    System.gc();
    ...
}
```

The requests to allocate memory chunks larger than a specific memory limit go through a code path that invokes a `System.gc()` call.

### **2.2.16 Issue 30**

The `allocateInstance` method of `com.google.apphosting.api.ReflectionUtils` class allows to allocate instances of arbitrary classes. It is implemented with the use of an `allocateInstance` method of `sun.misc.Unsafe` class, which takes one argument only denoting a class of an object to allocate.

The checks conducted prior to the unsafe object allocation operation primarily verify whether it is to be conducted for a class loaded by a user Class Loader. These checks are however insufficient. They allow for a call in a case of a NULL context Class Loader and also for the classes defined in prohibited packages (i.e. `sun.*`). This can be exploited to create valid instances of security sensitive classes such as `sun.misc.Unsafe` class.

### **2.2.17 Issue 31**

Issue 31 allows to obtain a reference to the `defineClass` method of `java.lang.ClassLoader` class through a Constant Pool of a Java Class file. GAE code inspects Constant Pool entries denoting certain Method Handles (`defineClass` methods), it does not however inspect the *EnclosingMethod* attributes, which can hold references to arbitrary instances of `java.lang.reflect.Method` class.

Issue 31 is a minor modification of a known Issue 63 we reported to IBM in May 2013 [15]. Oracle Java SE had a similar vulnerability, but the company fixed it by adding a security check to a `getEnclosingMethod` call (a check against `RuntimePermission("accessDeclaredMembers")`).

We initially classified Issue 31 as a manifestation of Issue 21 (old JRE), but due to the fact that GAE grants access to declared members for user applications, Issue 31 does not depend on the old JRE code base (it could be abused in GAE even with the most recent JRE 7U71).

## **2.3 AFFECTED COMPONENTS**

Discovered vulnerabilities had their origin in several improperly implemented GAE components. Fig. 18 shows specific security issues and their location (GAE components they originated from).

What can be seen from it is that a majority of issues originated from an insecure implementation of a GAE security layer, Class Sweeper and mirror classes in particular. These components were alone responsible for 20 vulnerabilities in total.

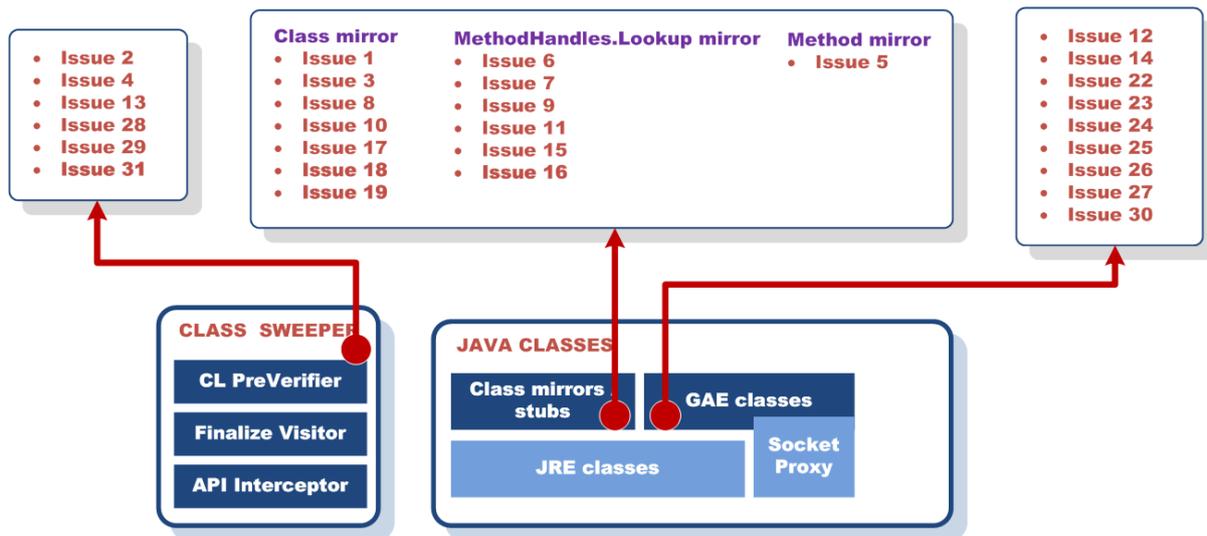


Fig. 18 Security issues and GAE components they originated from.

## 2.4 VENDOR'S EVALUATION

Google acknowledged that Security Explorations' report demonstrated that one of company's layers of defense had insufficient mitigations against certain type of attacks and the auditing of the privileged Java classes were insufficient.

The company provided a status report containing the results of its evaluation of the reported issues. Google concluded that the issues that worked<sup>24</sup> turned out to have as a root cause a common bug / class with a couple different exploitation vectors. The results of Google's root cause tracking / bugs evaluation are presented in Table 3.

Bug class	Status	Issues
URLClassLoader instantiation	ACCEPTED	1, 2, 4
unintercepted MethodHandles.Lookup.in(Class)	ACCEPTED	5
unintercepted methods in Lookup mirror	ACCEPTED	6, 7, 10, 11, 24
MethodHandles weaken original reflection model	ACCEPTED	8, 9, 12, 14, 15, 16, 22, 23, 25, 26, 27
findSystemClass() needs to be overwritten to check classes against whitelist	ACCEPTED	13
Reflection API doesn't disallow access to packages/members	WAI	17, 18
Class.getProtectionDomain() leak	WAI	19
runtime JAR files aren't protected against reading	WAI	20
outdated JRE	ACCEPTED	21
Insufficient checks in RuntimeVerifier.getApplicationCallerClass	ACCEPTED	3
UNKNOWN	ACCEPTED	28
UNKNOWN	UNKNOWN	29, 30, 31

Table 3 The results of Google's root cause tracking / bugs evaluation.

<sup>24</sup> this primarily concerns the issues reported after 12-Dec-2014 (after our access to the GAE environment was reenabled and all of the issues could be confirmed in a production).

This table indicates that Google treated some of the issues as not bugs, but *working as intended* (WAI) issues. They are described in a more detail below.

Issues with a bug class or status denoted as UNKNOWN didn't have a corresponding information provided by the company.

### **2.4.1 WAI issues**

Google evaluated Issues 17-20 as *working as intended issues*. The following arguments were used by the company to support its conclusion:

- For Issues 17 and 18 the company stated that it had a whitelist of classes, so it didn't consider this to be a security issue on its own. Overall, Google would consider that fixing these issues wouldn't provide a clear security barrier.
- For Issue 19, Google stated that if an attack requires instantiation of `URLClassLoader` or bypassing `defineClass()` interception and cannot be performed without these prerequisites then the above issues are considered to be the root causes. There is no expectation that the sandbox can function once the application has got hold of a real `URLClassLoader`.
- For Issue 20, the company agrees that this information can be used by an attacker to learn more about the JRE, however it would prefer not to depend on keeping this secret.

#### *2.4.1.1 Additional arguments*

In a response to the above, we provided additional arguments to Google regarding Issues 17-20. These are outlined below.

##### *Issue 17*

Oracle Java SE API documentation available at <http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html> specifies that `getMethod()` call of `java.lang.Class` class throws a `SecurityException` if a security manager, `SM`, is present and any of the following conditions is met:

- 1) invocation of `sm.checkMemberAccess(this, Member.PUBLIC)` denies access to the method,
- 2) the caller's class loader is not the same as or an ancestor of the class loader for the current class and invocation of `sm.checkPackageAccess()` denies access to the package of this class.

In a GAE environment, there is a security manager set. Condition 1 is not met, but condition 2 is fulfilled, but no `SecurityException` is thrown. Such an implementation violates Oracle's Java SE API.

Additionally, successful exploitation of Reflection API issues usually requires the following:

- access to an instance of a prohibited Class object (i.e. `sun.misc.Unsafe`),

- access to specific Method, Field or Constructor object of a prohibited class (i.e. `defineClass` method of `sun.misc.Unsafe` class).

By allowing reflection access to methods of classes denoted as prohibited by a Java security policy (in GAE, `package.access=sun.,\` etc.), the job is made easier for an attacker (one step less in an exploitation process).

Finally, if a deeper look into the code of a standard JRE `getMethod()` call (and the remaining Reflection API methods indicated by us) is made, the following implementation can be seen:

```
public transient Method getMethod(String s, Class aclass[])
    throws NoSuchMethodException, SecurityException {
    checkMemberAccess(0, Reflection.getCallerClass(), true);
    ...
}
```

There is a `checkMemberAccess()` call in the beginning of every Reflection API based method. And `checkMemberAccess()` invokes `checkPackageAccess()` among other things:

```
private void checkMemberAccess(int i, Class class1, boolean flag) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        ...
        checkPackageAccess(classloader, flag);
    }
}
```

GAE's Java API interception model lacks both of the above invocations. As a result, we conclude that it weakens the security model of a standard JRE<sup>25</sup>.

### *Issue 18*

Rationale for treating Issue 18 as a security bug is similar to Issue 17. Again, Oracle's Java SE API is violated (access to declared members is allowed, in a standard JRE security sandbox such an access is not allowed across class loader namespace boundaries).

Similarly to Issue 18, a standard JRE security model is also weakened and exploitation process made easier (i.e. certain declared static fields of whitelisted classes contain references to prohibited classes, security sensitive objects or methods).

In our POC codes, access to declared members is exploited in order to:

- obtain access to `sun.misc.Unsafe` Class instance (type of a declared field of `java.util.Random` class),
- obtain access to protected `defineClass` method of `java.lang.ClassLoader` class.

---

<sup>25</sup> Java Applets and Java Web Start applications, but also standalone Java applications with a Security Manager set.

If the above was not possible, whole exploit chains would be blocked as indicated in paragraph 3.2.

Finally, there is no rationale for giving unrestricted reflective access to all classes. Users can easily implement (expose) access to private fields of its own classes (from `UserClassLoader` namespace) by the means of standard Java language features. Users can also make use of Java SE 7 features (i.e. `MethodHandles`) for that purpose.

### *Issue 19*

This issue is an information leak. Security vulnerabilities of that type are usually helpful during the exploitation process. This is also the case here.

Unrestricted `getProtectionDomain()` call turned out to be useful when we initially approached GAE. It provided hints on which areas of Java security to focus on in order to break it.

A successful call to `getProtectionDomain()` also helped us gain access to:

- Permissions object assigned with user application classes (exploit *Scenario 1* described in 3.1.1)
- an instance of a `RuntimeClassLoader` object, which further opened access to `runtime-impl.jar` class namespace and associated vulnerabilities (Issues 23-27).

So, again we have a situation, where an innocent looking call turns out to be very helpful in a couple of exploitation scenarios. It also weakens a standard JRE security model (this call is not allowed in JRE due to rather sensitive nature of the object references it contains).

It's also worth to mention that the output of a `getProtectionDomain()` call encountered in Oracle Java Cloud environment did not leak that much information:

```
ProtectionDomain (
file:/customer/applications/Greeting/app/_WL_internal/servers/m0/tmp/_WL_us
er/Greeting/qb06jh/war/WEB-INF/lib/_wl_cls_gen.jar
<no signer certificates>
weblogic.utils.classloaders.ChangeAwareClassLoader@16bfb595
  finder:      weblogic.utils.classloaders.CodeGenClassFinder@16bfd510
  annotation:  Greeting@Greeting.war
<no principals>
java.security.Permissions@16f0f133 (
)
)
```

### *Issue 20*

This information leak issue was also helpful during the exploitation phase. We could in particular learn that `java.lang.ClassLoader` had some extra native calls added by Google to its implementation:

```
private static native long getLauncherHandle0();
private static native long findWithHandle0(long l, String s);
private static native void unloadLauncher0(long l);
```

The first two turned out to be in particular useful to retrieve addresses of `libc / libjavavruntime` symbols (for native code execution, inspection of certain memory structures, etc.).

We could also learn that GAE had 1+ years old JRE (Issue 21), which was helpful during privilege elevation phase (exploit vectors described in paragraph 3.1.2.1 and 3.1.2.2).

Class Loaders can be implemented in a way that does not expose anything from the classpath (system JAR content in particular), except from user provided classes / resources.

#### 2.4.1.2 Closing thoughts

As a response to the above, Google stated that it generally agreed that certain WAI Issues were deviations from a traditional Java security model. The company however indicated that it was necessary (dealing with web applications vs. java applets).

We did not agree with Google's evaluation methodology of reported issues. It is a common trend in attacks against technologies such as Java VM that more than one, partial and sometimes even innocent looking security issue needs to be combined together to achieve a serious security compromise. By focusing on the so called root cause, Google could easily miss an innocent vulnerability that may turn out to be helpful in a future attack. The company could also miss the opportunity to make a platform more secure by blocking certain issues that were part of real exploit chains.

Issue 5 is a good example of the risks associated with Google's evaluation methodology. This issue was originally reported as an instance of an "insecure use of `invoke` method of `java.lang.reflect.Method` class". Google concluded that a root cause for this vulnerability was the "unintercepted `MethodHandles.Lookup.in(Class)` call". As a result, an initial fix for Issue 5 deployed in GAE seemed to address the `in` method only. This is illustrated on Fig. 19.

```
java.lang.SecurityException: Unable to access class com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.Method_
at com.google.appengine.runtime.Request.process-d21eb9e1c8206839(Request.java)
at java.lang.invoke.MethodHandles$Lookup.in(MethodHandles$Lookup.java:38)
```

Fig. 19 Output of POC4 illustrating Issue 5 after an initial fix deployment.

That's equivalent to addressing an exploitation vector described in a paragraph 3.1.3.1 (abuse of an outer class implementation). As the fix didn't address the actual cause of Issue 5, upon some minor modification of POC4, a complete GAE security sandbox escape could be still achieved (new POC21 illustrating Issue 5).

### 3 EXPLOITATION TECHNIQUES

This paragraph provides information regarding exploitation of security issues described in a previous paragraph. Information about specific exploitation vectors and exploit chains making use of several vulnerabilities are presented.

#### 3.1 SPECIFIC EXPLOITATION VECTORS

In most cases, exploitation of the discovered vulnerabilities is straightforward. This in particular includes Issues 7 and 12, which are related to the acquiring and invocation of a `defineClass` method handle of `java.lang.ClassLoader` class. These issues can be exploited by defining a privileged class in an escape Class Loader namespace (the namespace, which is not a subject to the class sweeping)

There are some vulnerabilities requiring a specific exploitation vector, such as those that allow for arbitrary `URLClassLoader` instantiation (Issues 1, 2, 4 and 6) or Reflection API method invocation conducted from within a privileged CL namespace (Issues 5, 23-27). Below, a more detailed description is provided with respect to them.

##### 3.1.1 Generic privilege elevation scenarios

Several of our POC codes implement similar privilege elevation scenarios for a complete GAE security sandbox escape. These scenarios rely on several generic primitives that functionally correspond to specific Java SE API methods, of which some need to be invoked in a privileged context. These primitives and their corresponding API calls are presented in Table 4.

Primitive	Corresponding, functionally equivalent Java SE API
GET_DECLARED_FIELD	<code>getDeclaredField</code> method of <code>java.lang.Class</code> class invoked in a privileged context
SET_ACCESSIBLE	<code>setAccessible</code> method of <code>java.lang.reflect.AccessibleObject</code> class invoked in a privileged context
GET_FIELD_VALUE	<code>get</code> method of <code>java.lang.reflect.Field</code> class
SET_FIELD_VALUE	<code>set</code> method of <code>java.lang.reflect.Field</code> class
GET_METHOD	<code>getMethod</code> method of <code>java.lang.Class</code> class invoked in a privileged context
INVOKE_METHOD	<code>invoke</code> method of <code>java.lang.reflect.Method</code> class

Table 4 Generic primitives and corresponding Java SE API methods used in privilege elevation scenarios.

Our exploitation *Scenario 1* expressed with the use of the described generic primitives proceeds as following:

- 1) GET\_DECLARED\_FIELD primitive is used to obtain a private `permissions` Field object of `java.security.ProtectionDomain` class,
- 2) SET\_ACCESSIBLE primitive is used to override a security of the `permissions` Field obtained in step 1,
- 3) Protection Domain object associated with user defined classes is obtained with the use of Issue 19,

- 4) an instance of a `java.security.Permissions` object is created with an `AllPermission` permission added to it (all permissions set),
- 5) `SET_FIELD_VALUE` is used to assign the permissions field of a Protection Domain object obtained in step 3 with a value of a permissions set created in step 4.

What's interesting in the presented exploitation *Scenario 1* is that it does not turn security manager off. It is however completely sufficient to issue arbitrary method calls in a fully privileged scope. All that is required for it, is the invocation of a given method in a `doPrivileged` method block (enabling of the permissions associated with a user's class Protection Domain).

Our exploitation *Scenario 2* expressed with the use of the generic primitives proceeds as following:

- 1) `GET_DECLARED_FIELD` primitive is used to obtain a private, static `unsafe` Field of `java.util.Random` class,
- 2) `SET_ACCESSIBLE` primitive is used to override a security of the `unsafe` Field obtained in step 1,
- 3) `GET_FIELD_VALUE` primitive is used to read the value of the `unsafe` Field (an instance of `sun.misc.Unsafe` class),
- 4) `GET_METHOD` primitive is used to obtain access to a `defineClass` Method of `sun.misc.Unsafe` class (the class of object obtained in step 3),
- 5) `INVOKE_METHOD` primitive is used to call `defineClass` Method of `sun.misc.Unsafe` class and define a privileged `HelperClass` class in a system Class Loader namespace,
- 6) `HelperClass` class is instantiated and a Security Manager is turned off.

Exploitation *Scenario 2* makes use of more primitives and it does set the value of a Security Manager to NULL. It also relies on Issue 19 to obtain a reference to a Protection Domain object associated with user classes.

Both exploitation scenarios assume the presence of an escape Class Loader namespace. It is required in Scenario 1 for a successful invocation of a given method in a `doPrivileged` method block. Scenario 2 requires it for the execution of `GET_FIELD_VALUE`, `SET_FIELD_VALUE` and `INVOKE_METHOD` primitives. They are invoked on members of system classes - reflective access to such classes is prevented by a GAE Interception layer as indicated in paragraph 1.2.4.3.

### **3.1.2 URLClassLoader instance**

While, the ability to create a fully functional `URLClassLoader` instance can be used for an escape of a `UserClassLoader` namespace and all restrictions imposed on it (i.e. JRE Class Whitelist and Class Sweeper), it is not privileged enough to define user classes with arbitrary permissions. Thus, another exploitation vector is required to achieve a complete JRE sandbox escape.

This `URLClassLoader` instance is however able to load classes from restricted packages such as `sun.*`. That's due to the `loadClass()` method missing a proper security check<sup>26</sup>. When combined with Issue 17, arbitrary instances of these classes could be created and their methods called.

Below, descriptions of two `URLClassLoader` exploitation scenarios is provided that were used by us during the research of GAE security. Both of them involve conducting operations in two Class Loader namespaces. `UserClassLoader` namespace is used by default. All system method / method handle calls along with Issue 17 exploitation are conducted in it. `URLClassLoader` namespace is used to delegate invocation of Reflection API calls conducted on members from a non-user Class Loader namespace. As a result, GAE restrictions imposed on them could be bypassed.

### 3.1.2.1 `sun.swing.AccessibleMethod` (Oct 2012 exploit vector)

Our original POC code from Oct 2012 implementing a complete GAE security sandbox escape relied on Issue 1. It also made use of a `sun.swing.AccessibleMethod` class, which constituted an unpublished JRE exploit vector at that time. This exploit vector was in particular useful as the constructor of `AccessibleMethod` class contained `setAccessible` method invocation of `java.lang.reflect.AccessibleObject` class conducted in a `doPrivileged` method block. This is illustrated on Fig. 20.

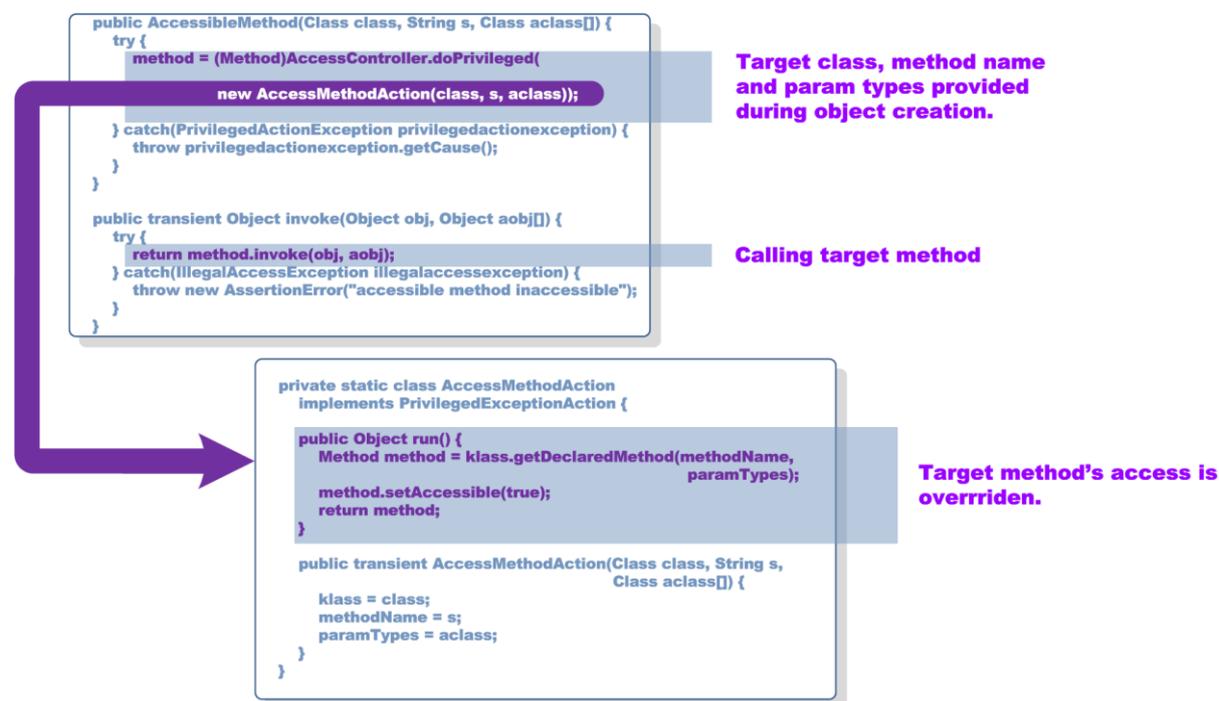


Fig. 20 The implementation of `sun.swing.AccessibleMethod` class.

This exploit vector could be used to override Java security protection for a given method denoted by a user. It could be any method, including private ones, of any system class.

<sup>26</sup> It's hard to blame Oracle for this as in JRE, pure instances of the `URLClassLoader` class are not used. Instead, objects of `java.net.FactoryURLClassLoader` class are used.

We exploited this condition to implement arbitrary privilege elevation of `UserClassLoader` classes with the use of exploitation *Scenario 1*. `GET_DECLARED_FIELD` primitive used by it was implemented with the use of Issue 18. `SET_ACCESSIBLE` primitive directly corresponded to the functionality of `sun.swing.AccessibleMethod` class.

### 3.1.2.2 `sun.swing.SwingLazyValue` (Oct 2014 exploit vector)

In our most recent POC codes we make use of `sun.swing.SwingLazyValue` class, which constitutes an old and less known JRE exploit vector (Issue 21). An instance of this class can be used as a proxy class calling methods of other classes through an insecure `invoke()` Reflection API call. This is illustrated on Fig. 21.

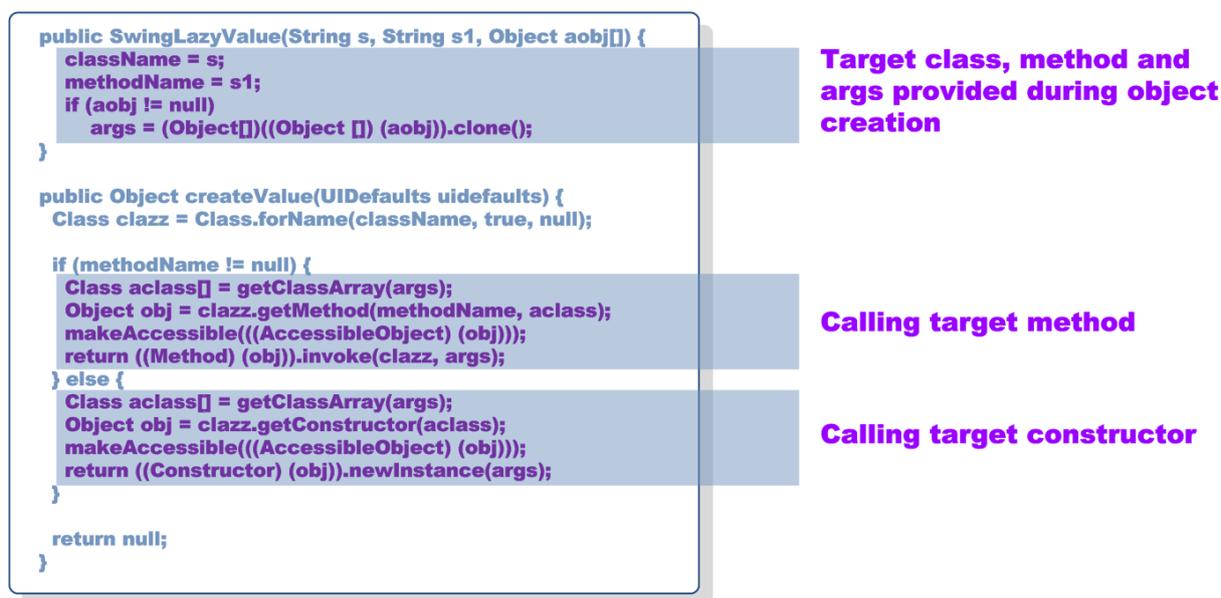


Fig. 21 The implementation of `sun.swing.SwingLazyValue` class.

Arbitrary method invocation conducted from within a system class can be abused by an attacker to implement access to its private members with the use of a new Reflection API. This exploitation technique is in particular valid for `sun.swing.SwingLazyValue` class as illustrated on Fig. 22.

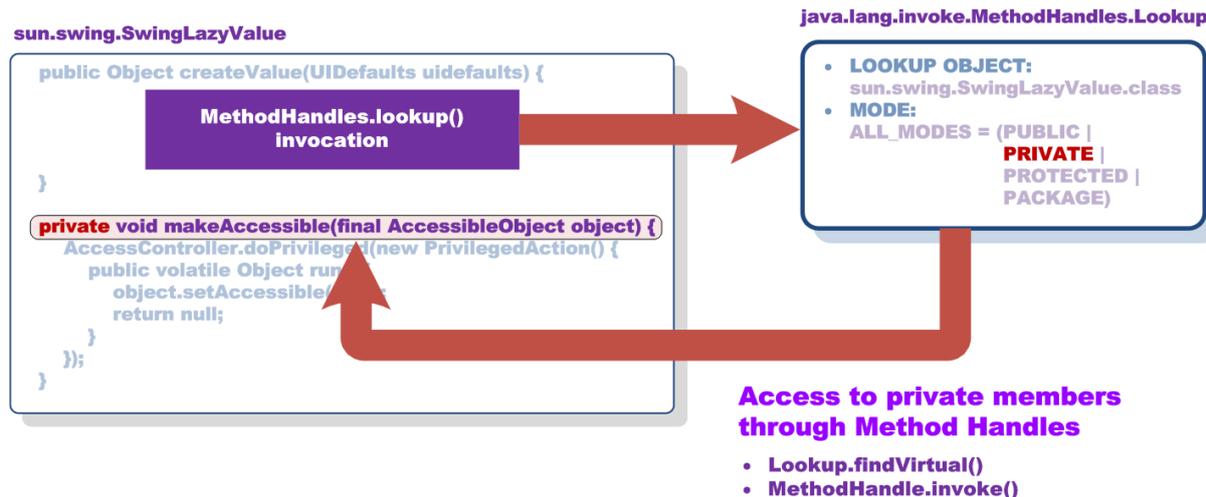


Fig. 22 The SwingLazyValue exploit vector.

In the new Reflection API, all reflective accesses to methods, constructors and fields are done with respect to the special lookup object, which is the instance of `MethodHandles.Lookup` class. This lookup object denotes the class with respect to which all method handle lookup operations are conducted. By default, this is the caller class of `MethodHandles.Lookup()` that is used as a lookup class.

An instance of `MethodHandles.Lookup` class created through the `invoke()` call embedded in a code of a `sun.swing.SwingLazyValue` class will have its lookup object set to the `SwingLazyValue` class itself. As a result, access to its private `makeAccessible` method could be gained. This method implements a privileged operation that overrides a security protection for a given `java.lang.AccessibleObject` class instance (i.e. Field or Method).

In our POC codes, a shared exploitation scenario is used for all `URLClassLoader` issues. It is implemented by a `EVector` class and implements exploitation *Scenario 2*. Again, `GET_DECLARED_FIELD` primitive used by it was implemented with the help of Issue 18. `SET_ACCESSIBLE` primitive directly corresponded to the functionality of the `makeAccessible` method of `sun.swing.AccessibleMethod` class. Issue 17 was used to implement the functionality of a `GET_METHOD` primitive. Finally, `GET_FIELD_VALUE` and `INVOKE_METHOD` primitives were implemented by corresponding Java SE APIs, but called in the `URLClassLoader` namespace.

### 3.1.3 *invoke() in a privileged CL namespace*

GAE code contained several instances of a classic Reflection API vulnerabilities originating from an insecure use of the `invoke` method of `java.lang.reflect.Method` class. Below, two exploitation scenarios are presented with respect to them that are in particular interesting.

### 3.1.3.1 Abuse of an outer class implementation (Issue 5)

Arbitrary method invocation conducted from within a more privileged class can be abused by an attacker to implement access to its private members with the use of a new Reflection API. This exploitation technique is in particular valid for Issue 5 as illustrated on Fig. 23.

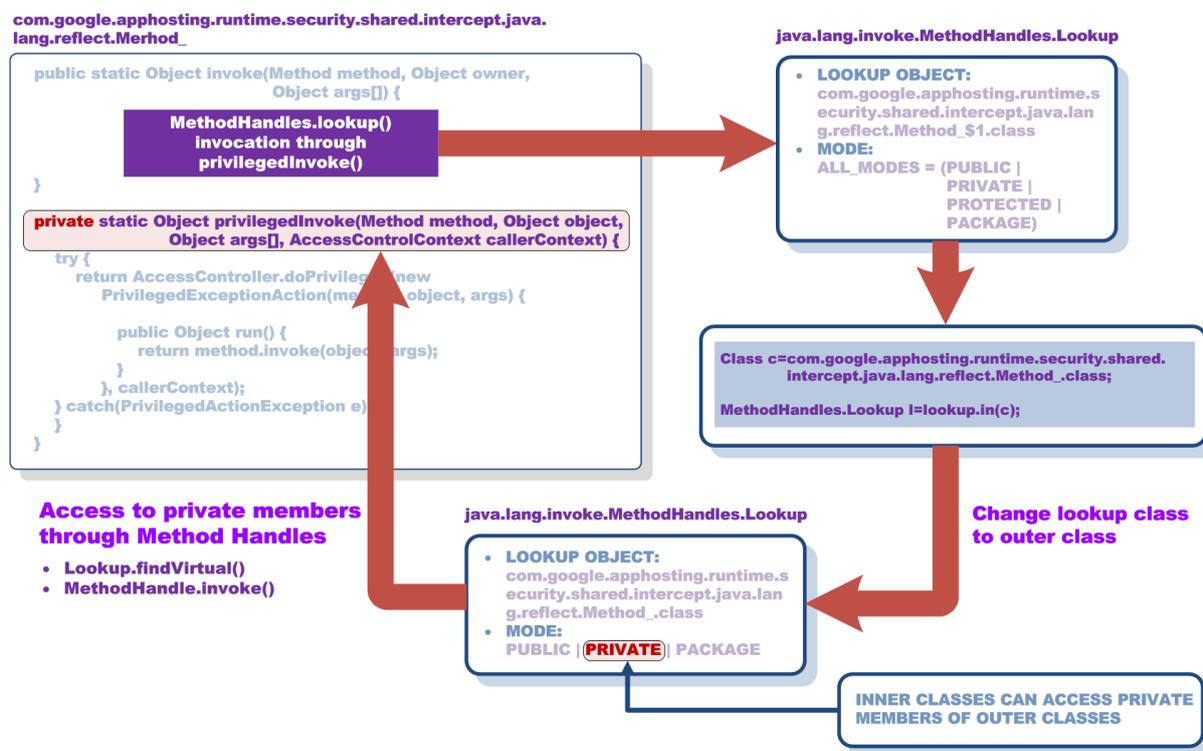


Fig. 23 The exploitation of Issue 5.

In our case, the `invoke` method is called from within a `doPrivileged` method block, thus arbitrary invocation of `MethodHandles.Lookup()` call would set `com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.Method_$1` class as a lookup object. Due to the fact that internal classes can access private members of the outer classes that enclose them, the lookup object can be changed to the outer class with the use of a `Lookup.in()` method call. This new lookup object can be abused to obtain method handles to either `setAccessiblePrivileged()` or `privilegedInvoke()` methods. The first method overrides a security protection for given `java.lang.AccessibleObject` class instance (i.e. `Field` or `Method`). The second one, allows for arbitrary method invocation inside a `doPrivileged` method block. The two method can be used in parallel to achieve a complete Java security sandbox escape with the use of exploitation *Scenario 2*. `GET_DECLARED_FIELD`, `GET_FIELD_VALUE`, `GET_METHOD` and `INVOKE_METHOD` primitives were implemented with the use of the abovementioned `privilegedInvoke` method. `SET_ACCESSIBLE` primitive directly corresponded to the functionality of the `setAccessiblePrivileged` method of `Method_` class.

### 3.1.3.2 *MethodHandleProxies implementing PrivilegedAction interface (Issues 23-27)*

Our Oracle Security Vulnerability report from Mar 2013 [18] described arbitrary JVM exploitation technique making use of the `asInterfaceInstance` method of `MethodHandleProxies` class. It relied on the possibility to create a `MethodHandleProxy` instance implementing a `java.security.PrivilegedAction` interface that executed a specially crafted method handle at the time of an interface method dispatch (method `run()` in this case). This specially crafted method handle was corresponding to `setSecurityManager` method of `java.lang.System` class with an argument bound to the NULL value. The idea behind the exploit was to provide a `MethodHandleProxy` instance as an argument to the `doPrivilegedWithCombiner` method call of `java.security.AccessController` class. As a result, a target method handle could be successfully executed with full privileges (in a privileged method block as all stack frames surrounding it were from privileged Class Loader namespaces).

Similar privilege elevation technique could be used in GAE for a successful exploitation of an arbitrary `invoke()` call done from within a privileged Class Loader namespace such as those corresponding to Issues 23-27. Below, a more detailed scenario is described with respect to them.

It should be noted, that presented exploitation scenario assumes an arbitrary escape of a `UserClassLoader` namespace. This is primarily due to the need to execute arbitrary user code in a finalizer thread (unrestricted `finalize()` methods).

#### *Issues 23, 25, 26 and 27*

These issues have their origin in classes defined in a `RuntimeClassLoader` namespace. Although this namespace is not fully privileged, it contains several privileges beyond those possessed by a `UserClassLoader` that can directly lead to a complete Java security sandbox escape. This in particular includes `"suppressAccessChecks"` and `"accessClassInPackage.sun.*"` runtime permissions.

In order to be able to create a `MethodHandleProxy` instance implementing a `java.security.PrivilegedAction` interface that could be used in the attack, the following two conditions need to be satisfied (Fig. 24):

- 1) a target method handle needs to be bound to a privileged<sup>27</sup> class,
- 2) a `MethodHandleProxy` instance needs to be created in a privileged Class Loader namespace.

---

<sup>27</sup> to a class, which privileges are to be exploited.

java.lang.invoke.MethodHandleProxies

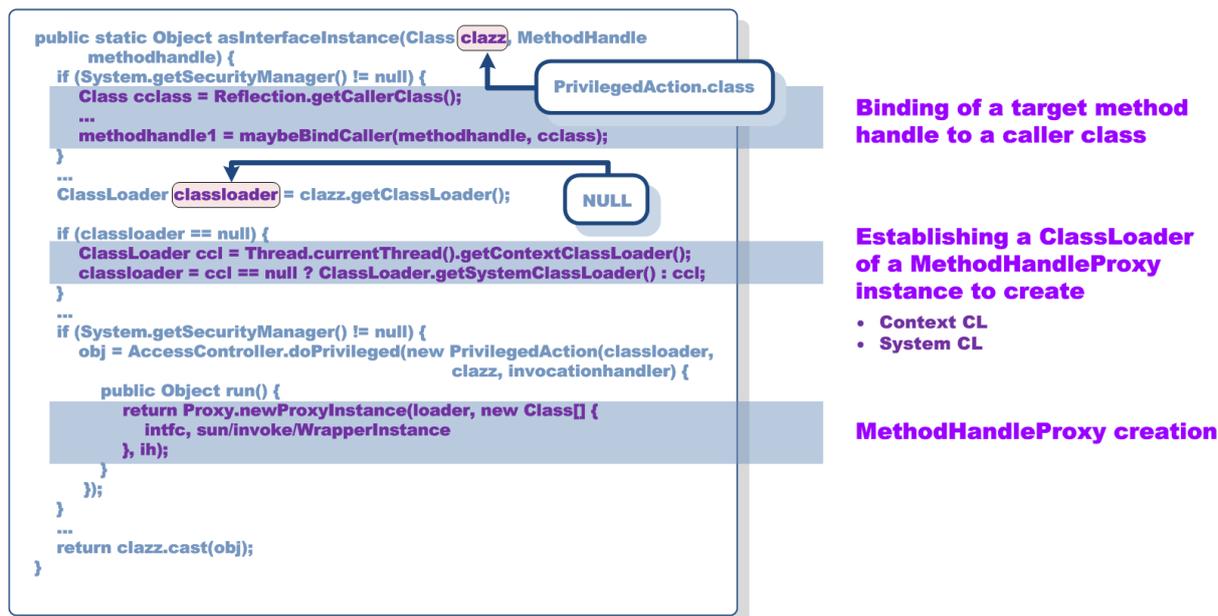


Fig. 24 MethodHandleProxy creation for java.security.PrivilegedAction interface.

For Issues 23-27, condition 1 is always satisfied as the caller of the asInterfaceInstance method of MethodHandleProxies class is a class from a RuntimeClassLoader namespace (caller of the insecure invoke method). Condition 2 can be met if either current Thread's getContextClassLoader() result yields a privileged Class Loader or a NULL value. In GAE, current Thread's getContextClassLoader() points to UserClassLoader by default, thus the only way to meet condition 2 is when it is set to the NULL value. This can be accomplished by issuing the asInterfaceInstance method call in a JVM's system finalizer thread (in any object's finalize() method). In such a case, a system Class Loader namespace will be used to define a MethodHandleProxy class. Thus, it needs to be privileged as well. In GAE, an instance of a system Class Loader has the following permissions (1.2.6):

```

sun.misc.Launcher$AppClassLoader@8d1800
<no principals>
java.security.Permissions@16782fa (
("java.lang.RuntimePermission" "exitVM")
("java.security.AllPermission" "<all permissions>" "<all actions>")
("java.io.FilePermission" "/base/java7_runtime/runtime-main.jar" "read")
)

```

The above indicates that its class loader namespace is fully privileged, which completes all conditions for the creation of a MethodHandleProxy instance implementing a java.security.PrivilegedAction interface.

In our POC codes, a helper method (run\_privileged method of InvokeHelper class) implementing arbitrary method invocation with a privileges of a Class Loader namespace embedding it is constructed as following (Fig. 25):

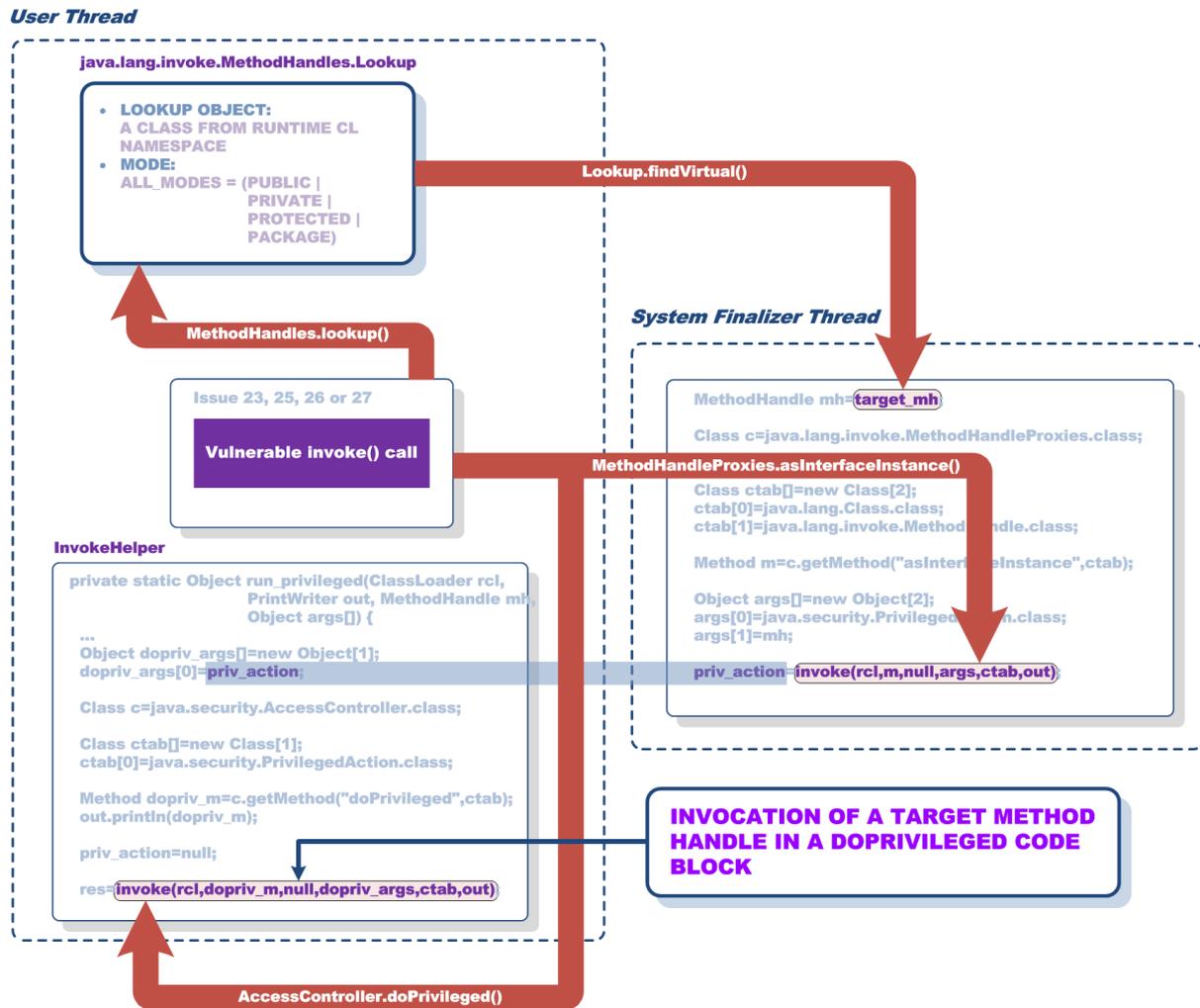


Fig. 25 An implementation of arbitrary method invocation with the privileges of a Class Loader namespace embedding a vulnerable invoke () call.

- the lookup method of MethodHandles class is invoked with the use of a given vulnerable invoke () call (Issue 23, 25, 26 or 27), as a result a privileged (ALL\_MODES) lookup object is obtained with a base lookup class denoting a class from a RuntimeClassLoader namespace (the caller of the lookup method corresponding to the class of the exploited issue),
- a target MethodHandle object (target\_mh) is obtained with the use of a findVirtual method of MethodHandles.Lookup class invoked on the lookup object acquired above,
- the asInterfaceInstance method of MethodHandleProxies class is invoked in a system finalizer thread with the use of a vulnerable invoke () call, a java.security.PrivilegedAction interface along with a target method handle are provided as method arguments, the call returns an instance of a MethodHandleProxy class (priv\_action),
- the doPrivileged method call of java.security.AccessController class is invoked with the use of a vulnerable invoke () call, the priv\_action instance obtained above is provided as an argument to it, as a result a target MethodHandle

object is invoked in a privileged method block (with the privileges of a Class Loader namespace embedding a vulnerable `invoke()` call).

A complete GAE security sandbox escape can be achieved with the use of exploitation *Scenario 2*. All primitives required by this scenario could be expressed with the use of a helper method described above (all security sensitive methods such as `getDeclaredField()`, `setAccessible()`, etc. can be invoked through the `run_privileged` method).

### Issue 24

Exploitation of Issue 24 can be also accomplished with the use of the technique described above. The `run_privileged` helper method can be however used to call any security sensitive method. The reason for it are the privileges of the vulnerable `com.google.apphosting.util.UserClassLoaderHelper` class. It comes from `jdk7_runtime-bootstrap.jar` code location, which is part of the fully privileged, system Class Loader namespace (1.2.6):

```
sun.boot.class.path=
  /base/java7_runtime/jdk7_runtime-bootstrap.jar:
  /base/jre7/lib/resources.jar:
  /base/jre7/lib/rt.jar:
  /base/jre7/lib/sunrsasign.jar:
  /base/jre7/lib/jsse.jar:
  /base/jre7/lib/jce.jar:
  /base/jre7/lib/charsets.jar:
  /base/jre7/classes
```

As a result, a constructed `MethodHandleProxy` will be always an instance of a fully privileged class and no restrictions will be imposed on `doPrivileged` method calls implemented by the `run_privileged` helper (all stack frames will be privileged). This is why only one method needs to be called through it in order to implement a complete GAE sandbox escape exploitation scenario. This is the `setSecurityManager` of `java.lang.System` class as illustrated on Fig. 26.

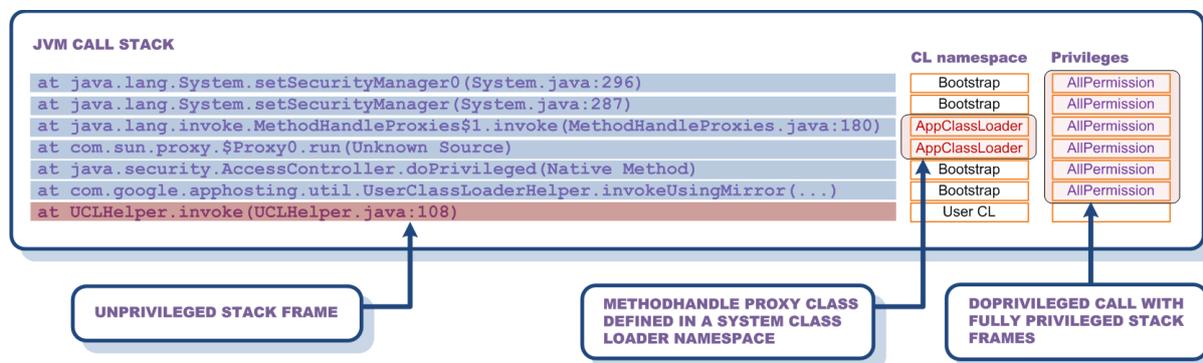


Fig. 26 JVM call stack during Issue 24 exploitation.

### 3.2 EXPLOIT CHAINS

Many of the discovered vulnerabilities needed to be chained together in order to create a successful GAE security sandbox bypass condition. Table 5 presents information about actual exploit chains used in our POC codes that lead to such a condition.

PoC #	Issue #																											
	1	2	3	4	5	6	7	8	9	10	11	12	14	17	18	19	21	22	23	24	25	26	27	28	29	30	31	
1	•													•				•										
2		•	•											•				•										
3				•										•				•										
4					•									•	•													
5						•								•				•										
6							•																					
7								•	•						•													
8										•	•				•													
9												•																
17													•			•			•									
18													•			•			•	•	•							
19						•										•			•									
20								•						•	•									•	•	•		
21				•																								
22									•																			•

Table 5 Exploit chains used in complete GAE security sandbox bypass POC codes.

It indicates that there are 27 vulnerabilities in total shared across 15 POC codes. Eight of them are completely independent full GAE escape codes (marked as red rows) making use of 17 different issues (indicated by yellow dots).

Additionally, Table 6 presents information about exploit chains relying on Issues 17-19 evaluated by Google as WAI Issues. It shows that 10 exploit chains would be broken (marked as red rows) if these issues (indicated by yellow dots) were not present in GAE.

PoC #	Issue #																											
	1	2	3	4	5	6	7	8	9	10	11	12	14	17	18	19	21	22	23	24	25	26	27	28	29	30	31	
1	•													•				•										
2		•	•											•				•										
3				•										•				•										
4					•									•	•													
5						•								•				•										
6							•																					
7								•	•						•													
8										•	•				•													
9												•																
17													•			•			•									
18													•			•			•	•	•							
19						•										•			•									
20								•						•	•									•	•	•		
21				•																								
22									•																			•

Table 6 Exploit chains relying on Issues 17-19.

A summary of the main exploit chains implemented by Proof of Concept codes accompanying this report is provided below:

- arbitrary `URLClassLoader` instantiation (via either Issue 1, 2, 4 or 6) is exploited through a restricted JRE class to which access is gained through Issues 17 and 21 (exploitation *Scenario 2*)
- acquired `defineClass` method of `java.lang.ClassLoader` class (Issue 8, 10 or 31) is turned into a method handle (Issue 9 or 11) in order to be able to call it (define a privileged class in an escape CL namespace),
- access to `RuntimeClassLoader` instance (via Issues 14 and 19) is combined with a Class Loader vulnerability (Issue 24) or a Reflection API bug (either Issue 23, 25, 26 or 27) in order to define a privileged class in an escape CL namespace or implement exploitation *Scenario 2* (via `MethodHandleProxies` implementing `PrivilegedAction` interface),
- invocation of a user provided code in a system finalizer thread (Issues 28 and 29) is chained with Issue 30 allowing for a restricted JRE class' instantiation, this is further exploited through Issues 17 and 21 (steps 4-6 of exploitation *Scenario 2*).

### 3.3 NATIVE CODE EXECUTION

A complete Java security sandbox escape can be exploited to gain access to both binary and class files implementing a GAE sandbox at Java VM level. A more detailed inspection of the environment requires access to the native OS layer though. This in particular includes an arbitrary memory reading and writing as well as a native code execution.

#### 3.3.1 Breaking type safety

As indicated in our SAT-TV research and SE-2012-01 report, core Reflection API can be easily abused to break Java type and memory safety [19][13] through a specially crafted manipulation of a `type` field value of a reflective Field object.

This deficiency of a Java Reflection API is exploited in our Proof of Concept codes as well. It forms a base for an implementation of an unsafe cast operation from `java.lang.Object` to `int`. Such an operation allows to convert Java VM references to arbitrary memory addresses. Itself, it constitutes a successful compromise of a Java type safety rules.

In our Proof of Concept codes, this is the `TCHelper` (type confusion helper class) class and its `getaddr` method that implements the abovementioned Reflection API abuse:

```
public static int getaddr(Object o)
```

#### 3.3.2 Breaking memory safety

While Java memory safety could be broken with the abovementioned Reflection API abuse, we decided to make use of the functionality of `sun.misc.Unsafe` class instead.

This class, among others, implements the following two native methods:

```
public native int getInt(long l);  
public native void putInt(long l, int i);
```

They can be used to read and write arbitrary `int` values from a memory address denoted by their first argument. When combined with the `getaddr` method, both methods could be used to read and write values of Java objects or classes regardless of their security access.

In our Proof of Concept codes, these are the `read_mem` and `write_mem` methods of the `API` class that form a base for all memory access related operations. They wrap around the abovementioned methods of `sun.misc.Unsafe` class.

### 3.3.3 Gaining code execution

Arbitrary native code execution is achieved by exploiting the fact that GAE JVM is based on a HotSpot JVM for which optimization and JIT compilation of most frequently used (hotspots) Java bytecode sequences lie at the core of its functionality.

#### 3.3.3.1 *methodOop's adapter handle*

Support for JIT means that JVM both allocates and maintains dedicated memory areas (pools) that contain native code generated at runtime. Some of these areas have read, write and execute memory permissions set, which makes them a perfect target for an overwrite with an arbitrary user provided code to execute. This in particular includes the `_c2i_unverified_entry` memory area. Its address can be retrieved by navigating the internal JVM Class representation (*instanceKlass* structure) and its methods table. This is illustrated on Fig. 27.

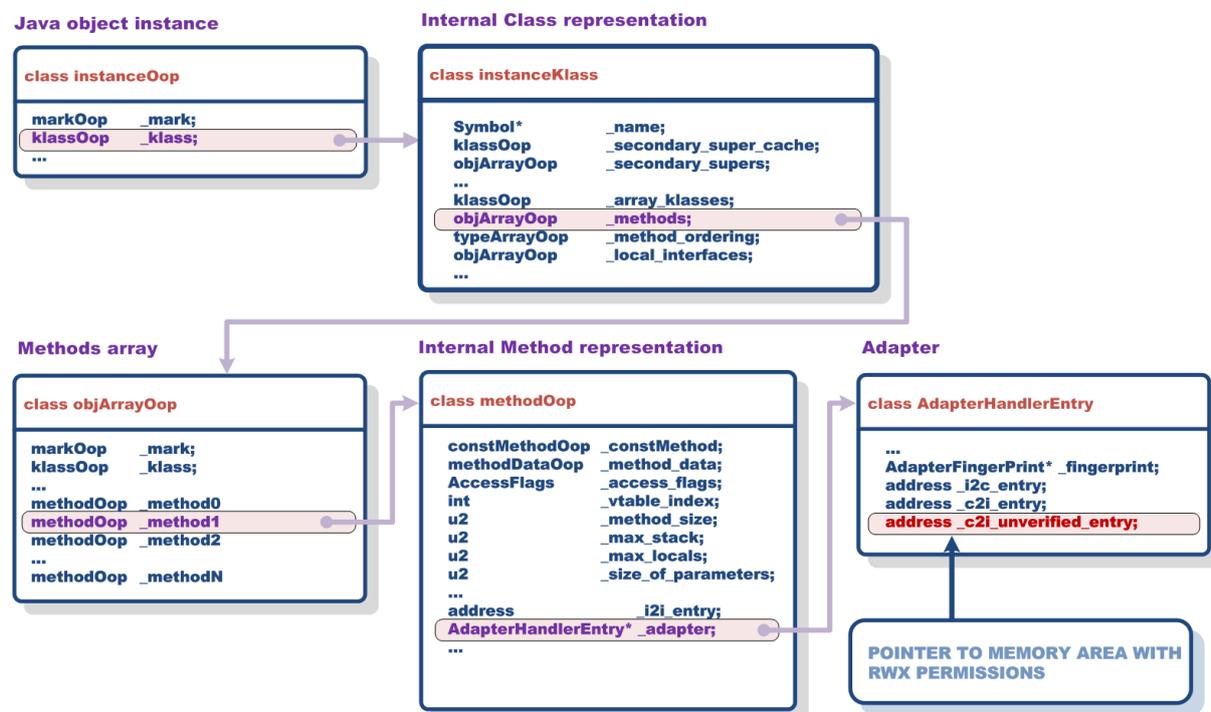


Fig. 27 Discovering the rwx memory area pointed by *methodOop's* adapter handle.

The *MethodOop's* structure corresponds to the internal representation of a class' method. It contains a pointer to an adapter handle encompassing several marshalling adapters responsible for a setup (i.e. arguments marshalling, frames setup) and invocation of

arbitrary code transfers in JVM. This in particular includes an invocation of an interpreted call from a compiled code, which is handled by the `_c2i_unverified_entry` adapter.

In our Proof of Concept codes, the memory area pointed by the `_c2i_unverified_entry` adapter is used as an initial storage for a user provided native code instructions to execute.

### 3.3.3.2 NativeSignalHandler

A native `handle0` method of `sun.misc.NativeSignalHandler` class is abused to achieve an arbitrary code execution dispatch from a given memory address. It takes two arguments denoting a signal number and a corresponding code handler:

```
private static native void handle0(int signum, long handle);
```

The implementation of the `handle0` method looks as following:

```
Java_sun_misc_NativeSignalHandler_handle0 proc near
    push    ebp
    mov     ebp, esp
    sub     esp, 18h
    mov     dword ptr [esp+8], 0
    mov     dword ptr [esp+4], 0
    mov     eax, [ebp+10h]
    mov     [esp], eax
    call   [ebp+14h]
    leave
    retn

Java_sun_misc_NativeSignalHandler_handle0 endp
```

Since this is a native method, its arguments need to follow the Java Native Interface Specification [20]. This implicates the meaning of the arguments upon native `Java_sun_misc_NativeSignalHandler_handle0` function entry as illustrated in Table 7.

Argument	Meaning
<b>[ebp+08h]</b>	JNIEnv ptr
<b>[ebp+0ch]</b>	this ptr
<b>[ebp+10h]</b>	arg0 (signum)
<b>[ebp+14h]</b>	arg1 (code handler)

Table 7 Arguments' meaning of a native `Java_sun_misc_NativeSignalHandler_handle0` function.

The meaning of the arguments indicates that the invocation of the `handle0` method with a `handle` argument denoting a given memory address will start executing code from it. Such an execution will be done with a `signum` value passed as a first argument (on top of the stack).

### 3.3.3.3 Generic code\_handle

In order to be able to invoke arbitrary library calls through the `NativeSignalHandler` class, our POC codes make use of a generic `handle_code`, which wraps arbitrary native code invocations:

```

push    ebp
mov     ebp,esp

push    ebx           ;save regs
push    ecx
push    edx
push    esi
push    edi
sub     esp,028h      ;alloc tmp space for args

cld
mov     esi,[ebp+0x08] ;ptr to args area (signum)
mov     eax,[esi]      ;target code addr to invoke
add     esi,0x04       ;addr of first arg
mov     edi,esp        ;copy args to stack
mov     ecx,08h
rep     movsd
call   eax             ;call target function
mov     esi,[ebp+0x08] ;ptr to args area (signum)
mov     [esi],eax      ;store a function result

add     esp,028h      ;restore stack ptr
pop     edi           ;restore regs
pop     esi
pop     edx
pop     ecx
pop     ebx
pop     ebp
retn

```

The `handle_code` treats the first argument provided (`signum`) as a pointer denoting the memory area holding the values of a target code address to invoke (offset `0x00`) and its arguments (offset `0x04`). Upon setting up a local stack frame and filling it with the arguments passed, a target code gets invoked through an indirect `call` instruction. Upon completion of a called function, its returned value is stored back into the memory location denoted by a `signum` argument.

#### 3.3.3.4 Native code execution setup

In our Proof of Concept codes, the code pointed by the `_c2i_unverified_entry` adapter is overwritten with a `handle_code` sequence that invokes an `mprotect` libc call. As a result, the permissions for a given memory region (`rw_x_chunk`) provided as an argument to the call are changed, so that arbitrary code could be executed from it.

The `_c2i_unverified_entry` is overwritten only for the time of a `mprotect` invocation. Its content is restored, once the `mprotect` completes its job and `rw_x_chunk` is ready for use. The value of the `rw_x_chunk` itself is obtained through the `malloc` primitive described in the next paragraph.

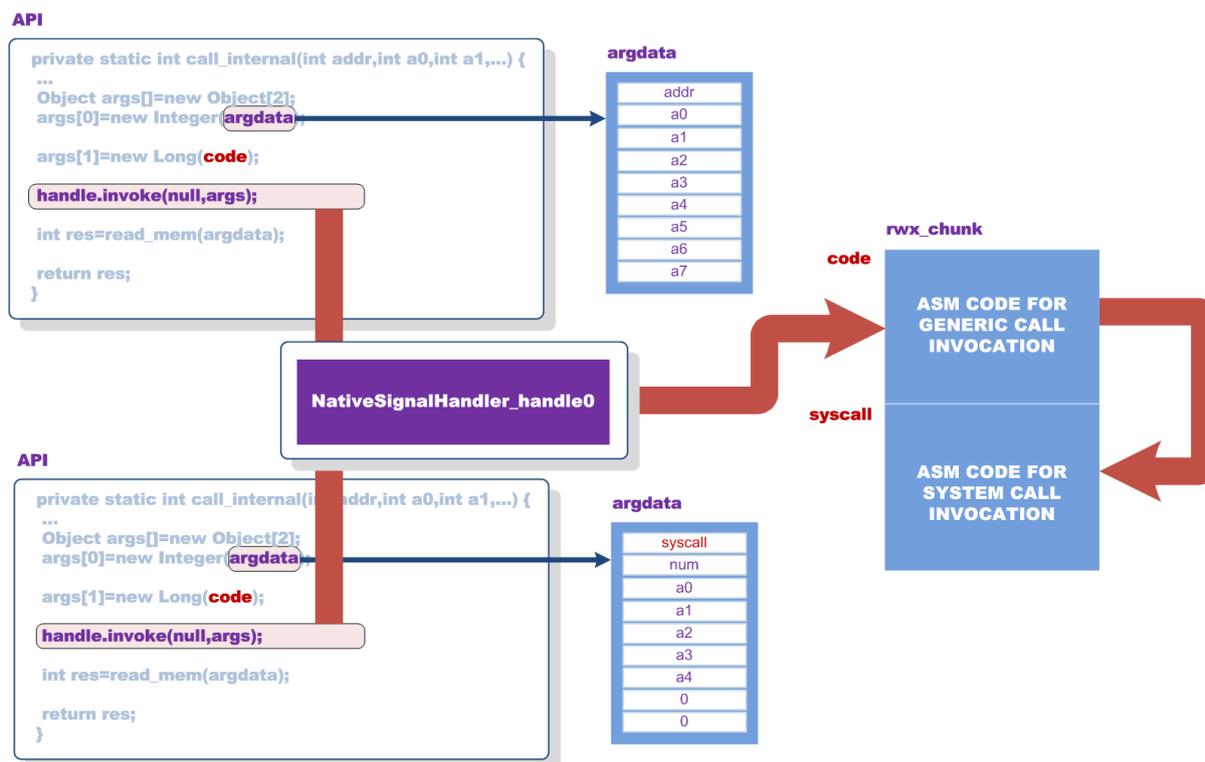


Fig. 28 The native code execution setup.

Upon changing the permissions of `rwx_chunk` to `PROT_READ|PROT_WRITE|PROT_EXEC`, generic code sequences implementing arbitrary library (`handle_code`) and system call (`syscall_code`) invocations are copied to it.

From that moment, these invocation wrappers can be used through the native `handle0` method of `sun.misc.NativeSignalHandler` class. All that's needed for that purpose is a proper setup of the `signum` argument. For a library call, it needs to point to the area holding a library address and its arguments. For a system call, it needs to hold a `syscall` number and corresponding arguments.

The described native code execution setup is illustrated in Fig. 28.

It is worth to mention that the described native code execution functionality is fully reliable. It is also capable to bypass certain exploit mitigation techniques such as Address Space Layout Randomization (ASLR) and (Data Execution Prevention) DEP that are supported by the underlying operating system.

### 3.3.4 Native API

For the purpose of an easier native layer access, the `API` class was developed that implements several methods for an arbitrary memory access and native code execution in particular. These methods are briefly described below.

### 3.3.4.1 Arbitrary memory access

As indicated in 3.3.2, the base primitives for arbitrary memory read and write operations are implemented with the use of `getInt` and `putInt` methods of `sun.misc.Unsafe` class. They are exposed as the following API methods:

- `int read_mem(int addr)`
- `void write_mem(int addr, int val)`

### 3.3.4.2 Native code execution

The API class implements two methods that allow for arbitrary native code execution:

- `int call(int addr, int a0, int a1, int a2, int a3, int a4, int a5, int a6, int a7)`
- `int syscall(int num, int a0, int a1, int a2, int a3, int a4)`

The `call` method invokes a function at a given memory location and with a list of arguments provided. The `syscall` method is similar - it invokes a system call denoted by a given number and arguments.

### 3.3.4.3 Symbol address lookup

As indicated in 2.4.1.1, Google added `getLauncherHandle0` and `findWithHandle0` native methods to the implementation of `java.lang.ClassLoader` class. These methods allow to find native addresses of the symbols exported by the Java VM runtime binary in particular. They are functionally similar to `dlopen` and `dlsym` Unix dynamic linker functions.

The handle returned by the invocation of the first method always corresponds to the main program. This is due to the NULL library name argument that this function passes to the underlying native `load` method of `ClassLoader.NativeLibrary` class.

The API class wraps the abovementioned native `ClassLoader` methods into a single method:

- `int get_addr(final String name)`

### 3.3.4.4 Malloc and free primitives

The API class implements two methods that allow for arbitrary allocation and freeing of memory chunks:

- `int malloc(final int size)`
- `void free(final int addr)`

They are implemented with the use of `allocateMemory` and `freeMemory` methods of `sun.misc.Unsafe` class.

### 3.3.4.5 Sample uses

The `API` class makes it possible to call native code from Java as if it was yet another Java method. A code sequence implementing arbitrary invocation of a `getpid()` libc function call is presented below:

```
int getpid=API.get_addr("getpid");
out.println("getpid: "+Integer.toHexString(getpid));
int res=API.call(getpid,0,0,0,0,0,0,0,0);
out.println("getpid res: "+Integer.toHexString(res));
```

It produces the following output:

```
getpid: f49876b0
getpid res: 1476
```

Additionally, the `API` class contains several additional helper methods that among other things allow to read C strings or blocks of data from a given memory location. A code sequence implementing arbitrary read of the contents of the program's `argc` and `argv` variables is provided below:

```
int addr=API.get_addr("__google_auxv");

int argc=API.read_mem(addr-0x0c);
int argv=API.read_mem(addr-0x08);
int envp=API.read_mem(addr-0x04);

out.println("argc: "+Integer.toHexString(argc));
out.println("argv: "+Integer.toHexString(argv));

for(int i=0;i<argc;i++) {
    addr=API.read_mem(argv+i*4);
    String s=API.read_string(addr);
    out.println("argv["+i+"] = "+s);
}
```

The following output is produced by it:

```
argc: 34
argv: ff82db14
argv[0] = /base/java7_runtime/java_runtime_launcher-piii-linuxopt
argv[1] = --appengine_release_name=1.9.16
argv[2] = --java_soft_deadline_ms=10600
argv[3] = --java_hard_deadline_ms=10200
argv[4] = *INTENTIONALLY REMOVED*
argv[5] = --external_datacenter_name=us2
argv[6] = --jvm_flags=-Xms32m
argv[7] = --enable_gae_cloud_sql_jdbc_connectivity
argv[8] = --interrupt_threads_first_on_soft_deadline
argv[9] = *INTENTIONALLY REMOVED*
argv[10] = *INTENTIONALLY REMOVED*
argv[11] = --application_root=/base/data/home/apps
argv[12] = --port=-1
```

```
argv[13] = --api_call_deadline=5.000000
argv[14] = --max_api_call_deadline=10.000000
...
```

The `API` class turned out to be in particular useful during the reverse engineering of a Java VM runtime binary (`libjavavaruntime.so`) and its runtime behavior.

## 4 VULNERABILITIES IMPACT

Successful exploitation of the vulnerabilities could allow to bypass GAE whitelisting of JRE classes and achieve a complete Java VM security sandbox escape. As a result, access to the files (binary / classes) comprising the JRE sandbox could be gained. By breaking Java memory safety, arbitrary native code execution could be also achieved in a target GAE environment (ability to issue arbitrary library / system calls).

While, we haven't reached a point in our research where we could state that arbitrary compromise of other GAE user's data or applications is possible (bypass of the first sandboxing layer of App Engine, with the remaining layers intact), the achieved security compromise did constitute a considerable information leak. It could be used to gain a lot of information about the JRE sandbox itself, Google internal services and protocols. It also seemed to be a potentially good starting point to proceed with attacks against the OS sandbox and RPC services visible to the sandboxed Java environment.

Below, more details are provided with respect to the information leak itself. The following information could be gained upon a successful compromise of a GAE security sandbox:

- binary and Java codes implementing the GAE JVM runtime, that include the monster `libjavavaruntime.so` binary (468416808 bytes) and `runtime-impl.jar` archive of Java classes (121611977 bytes) in particular,
- full DWARF debug information included in binary files (type information and such),
- PROTOBUF [21] definitions from Java classes (description of 57 services in 542 .proto files),
- PROTOBUF definition from binary files (description of 8 services in 68 .proto files),
- many URLs denoting Google source code repositories and corporate web addresses left in code,
- static configuration data for Google services (355 services in total).

PROTOBUF definitions mentioned above needed to be extracted from binary / class files. We used a small tool (*ExtractProto*) for that purpose that generated proper ASCII representation of the available protocols definitions. The contents of APPENDIX B was obtained with the use of this tool.

It should be also mentioned, that PROTOBUF definitions did constitute a significant information leak in particular. They included information about internal Google services of which many appeared rather unrelated to Google App Engine (i.e. Android, PartnerServices, GAIA auth / security stuff). The PROTOBUF definitions carried information about protocols,

their dependencies and services' definitions in a form of specific request / response messages.

Finally, binary codes implementing GAE runtime were not properly built. Apart from significant debugging information left in it, many client side code was accompanied by a code that seemed to be a part of a server end. This in particular includes the OS Sandbox related components. Similarly, GAE runtime classes included huge amounts of code implementing Google's sensitive functionality and protocols related to security and authentication, monitoring, file systems and ads in particular. In general, this all looked as if all core internal Google APIs and libraries were incorporated into the GAE runtime. That's likely because Google GAE integrates tightly with a core, RPC service based middleware layer on top of which all other, internal Google services run (all Google runs ?).

## 5 SUMMARY

Securing cloud based environments that allow for arbitrary deployment and execution of user provided code is a challenging task. Current solutions are usually built upon a specific sandboxing mechanism, either custom built or implemented with the use of a virtualization.

In case of Google App Engine for Java, its first layer of defense was built around a Java VM sandbox. Google decided to implement an additional security layer (sandbox) on top of it. As a result, several custom security measures were integrated into a Java VM runtime. This in particular concerns the Class Sweeper, of which goal was to verify and transform untrusted user code into a corresponding, safer representation. The API Interception and Interjection mechanism was meant to enforce proper security checks in runtime for security sensitive Java SE API calls. Finally, the JRE Class Whitelisting was supposed to limit the scope of Java classes visible to user applications.

Unfortunately, the custom security layer implemented by Google turned out to be vulnerable to multiple security weaknesses. Some of them were instances of known vulnerabilities published in the past. This in particular concerns the issues disclosed as part of SE-2012-01 research affecting Java SE implementation from Oracle and IBM (Issues 4, 9 and 31). This also concerns several security vulnerabilities that made it possible to break Oracle Java Cloud Service (Issues 1 and 4). The ability to break GAE with the use of a prior research indicates that it was either ignored or simply never taken into account.

The majority of the flaws discovered in GAE were related to either Reflection API or Class Loaders. These were rather simple issues, which should have been caught during a security review process preceding a release of GAE software (or any major update to it).

Reflection API and Class Loaders are fairly complex and security sensitive components of a Java VM. A lot of expert knowledge and a deep understanding of their operation is usually required prior to introducing any changes to them without jeopardizing the security of a JVM. Regardless of that, Google decided to "reimplement" Java Reflection API, through the GAE interception layer. This was the same API that caused so much trouble for Oracle in the recent years and that was responsible for dozens of security issues in Java. The company

also allowed for a creation of arbitrary user provided Class Loaders in GAE, which immediately created a need to protect the environment from these objects.

As a result, a security model of a standard JRE was weakened (Issue 17-19). Arbitrary vulnerabilities were introduced on top of the implementation of JRE API calls (Issues 3 and 5), which GAE Java API interception model aimed to protect. These vulnerabilities constituted the same violations of Java Secure Coding Guidelines [22] of which Oracle has been usually accused of. Many vulnerabilities had its origin in an incomplete interception of Java SE 7 method handles API (Issues 7, 9, 11, 15, 16), Class Loader's operation (Issue 13) or mitigations aimed at making user Class Loaders less privileged than usual (Issues 8, 10, 12, 14).

During our correspondence with Google, the company often emphasized that we only broke the first layer of defense and that it considered the remaining, lower sandboxing layers sufficiently robust. This could explain why the environment of a cloud computing platform from Google ran on a 1+ year old Java runtime (Issue 21). However, the amount and nature of information leaked by the first sandboxing layer along with the company's preference not to have the details of the next sandboxing layers published seemed to contradict the confidence expressed regarding their robustness.

It should be also mentioned that regardless of a successful detection of our activity in GAE, Google's ability to detect attacks in the environment was not perfect. Our activity raised an alarm 2 years after an initial GAE security sandbox compromise. It was likely detected because we decided to launch more aggressive (more visible / risky) tests and did not follow our usual, low-profile pattern of activity.

Google is a specific software vendor that serves hundreds of millions of users on a daily basis through its custom services. In most cases, the architecture and implementation details of these services are not known due to their server-side nature. As a result, the ability to discover security issues in these services could be quite challenging. Without any doubt security of Google services is not less important than discovering vulnerabilities in a client / server side software of other big software vendors. That thought alone should catch attention of Google itself. At the end of a day, it might turn out that it would be of a more benefit to the company and users of its services to have Google security personnel to be more focused on its own products instead of the products of the competition [23]. The case of Google App Engine for Java shows that this might actually make sense and that there are still places for improvement in Google's own yard.

## REFERENCES

- [1] SE-2012-01 Security vulnerabilities in Java SE  
<http://www.security-explorations.com/en/SE-2012-01.html>
- [2] Google App Engine: Platform as a Service  
<https://cloud.google.com/appengine/docs>
- [3] [SE-2014-02] Google App Engine Java security sandbox bypasses (project pending completion / action from Google)  
<http://seclists.org/fulldisclosure/2014/Dec/26>

**[4] Google Security Research**

<http://code.google.com/p/google-security-research/>

**[5] Mac Flashback trojan exploits unpatched Java vulnerability, no password needed**

<http://arstechnica.com/apple/2012/04/mac-trojan-exploits-unpatched-java-vulnerability-no-password-needed/>

**[6] Disable Java NOW, users told, as 0-day exploit hits web**

[http://www.theregister.co.uk/2012/08/27/disable\\_java\\_to\\_block\\_exploit/](http://www.theregister.co.uk/2012/08/27/disable_java_to_block_exploit/)

**[7] Java Runtime Environment**

<https://cloud.google.com/appengine/docs/java/>

**[8] URL Fetch Java API Overview**

<https://cloud.google.com/appengine/docs/java/urlfetch/>

**[9] ASM - Java bytecode manipulation and analysis framework**

<http://asm.ow2.org/>

**[10] The Java Virtual Machine Specification, Java SE 7 Edition**

<http://docs.oracle.com/javase/specs/jvms/se7/html/>

**[11] Java and Java VM security vulnerabilities and their exploitation techniques, Last Stage of Delirium Research Group**

<http://lsd-pl.net/>

**[12] SE-2013-01-ORACLE, Issues #1-28**

<http://www.security-explorations.com/materials/SE-2013-01-ORACLE.pdf>

**[13] Security Vulnerabilities in Java SE, technical report**

<http://www.security-explorations.com/materials/se-2012-01-report.pdf>

**[14] [SE-2012-01] 'Fix' for Issue 32 exploited by new Java 0-day code**

<http://seclists.org/fulldisclosure/2013/Jan/66>

**[15] SE-2012-01-IBM-2, Issue #62-68**

<http://www.security-explorations.com/materials/SE-2012-01-IBM-2.pdf>

**[16] Java version history, Wikipedia**

[http://en.wikipedia.org/wiki/Java\\_version\\_history](http://en.wikipedia.org/wiki/Java_version_history)

**[17] SE-2012-01-ORACLE-13, Issue #69**

<http://www.security-explorations.com/materials/SE-2012-01-ORACLE-13.pdf>

**[18] SE-2012-01-ORACLE-11, Issue #56-60**

<http://www.security-explorations.com/materials/SE-2012-01-ORACLE-11.pdf>

**[19] Security threats in the world of digital satellite television, Hack in the Box Security Conference, Amsterdam 2012**

<http://www.security-explorations.com/materials/se-2011-01-hitb1.pdf>

**[20] Java Native Interface Specification**

<http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>

**[21] Protocol Buffers - Google's data interchange format**

<http://code.google.com/p/protobuf/>

**[22] Secure Coding Guidelines for Java SE**

<http://www.oracle.com/technetwork/java/seccodeguide-139067.html>

**[23] Project Zero, <http://googleprojectzero.blogspot.com/>**

## APPENDIX A

### SUMMARY OF THE VULNERABILITIES

ISSUE #	TECHNICAL DETAILS	
1	origin	com.google.apphosting.runtime.security.shared.interject.java.lang.Class class com.google.apphosting.runtime.security.shared.intercept.java.lang.Class class
	cause	getConstructor()/newInstance() methods of java.lang.Class mirror missing Class Loader instantiation checks
	impact	arbitrary system Class Loader instantiation (i.e. java.net.URLClassLoader)
	type	partial GAE security bypass vulnerability
2	origin	Class Sweeper
	cause	java.security.Provider.Service is a whitelisted / not mirrored class
	impact	arbitrary system Class Loader instantiation (i.e. java.net.URLClassLoader)
	type	partial GAE security bypass vulnerability
3	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.Class class
	cause	insecure use of forName() method of java.lang.Class class
	impact	arbitrary access to restricted classes
	type	partial security bypass vulnerability
4	origin	Class Sweeper
	cause	java.beans.XMLDecoder is a whitelisted / not mirrored class
	impact	arbitrary system Class Loader instantiation (i.e. java.net.URLClassLoader)
	type	partial GAE security bypass vulnerability
5	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.Method class
	cause	insecure use of invoke() method of java.lang.reflect.Method class
	impact	arbitrary invocation of methods with user provided arguments
	type	complete GAE security bypass vulnerability
6	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.MethodHandles.Lookup class
	cause	no interception of findConstructor() method of java.lang.reflect.MethodHandles.Lookup mirror
	impact	arbitrary system Class Loader instantiation (i.e. java.net.URLClassLoader)
	type	partial GAE security bypass vulnerability
7	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.MethodHandles.Lookup class
	cause	no interception of findSpecial() method of java.lang.reflect.MethodHandles.Lookup mirror
	impact	java.lang.ClassLoader's defineClass access through special MethodHandle
	type	complete GAE security bypass vulnerability
8	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.Class class
	cause	missing defineClass(String, ByteBuffer, ProtectionDomain) call in

		defineClassOverloads map
	impact	access to security sensitive defineClass method of java.lang.ClassLoader class
	type	partial GAE security bypass vulnerability
9	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.MethodHandles.Lookup class
	cause	no interception of unreflect() method of java.lang.reflect.MethodHandles.Lookup mirror
	impact	java.lang.ClassLoader's defineClass access through MethodHandle
	type	partial GAE security bypass vulnerability
10	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.Class class
	cause	unfiltered getDeclaredMethods() call of java.lang.Class mirror
	impact	access to security sensitive defineClass method of java.lang.ClassLoader class
	type	partial GAE security bypass vulnerability
11	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.MethodHandles.Lookup class
	cause	no interception of unreflectSpecial() method of java.lang.reflect.MethodHandles.Lookup mirror
	impact	java.lang.ClassLoader's defineClass access through MethodHandle
	type	partial GAE security bypass vulnerability
12	origin	com.google.apphosting.runtime.security.shared.SafeClassDefiner class
	cause	missing safeDefineClass(ClassLoader, String, ByteBuffer, ProtectionDomain) method in SafeClassDefiner implementation
	impact	access to security sensitive defineClass method handle of java.lang.ClassLoader class
	type	complete GAE security bypass vulnerability
13	origin	Class Sweeper
	cause	no handling of findSystemClass() method of java.lang.ClassLoader class by Class Loader PreVerifier
	impact	arbitrary loading of system classes (whitelisting escape)
	type	partial GAE security bypass vulnerability
14	origin	com.google.apphosting.runtime.security.shared.SafeClassDefiner class
	cause	missing safeDefineClass(ClassLoader, byte[], int, int) method in SafeClassDefiner implementation
	impact	access to security sensitive defineClass method handle of java.lang.ClassLoader class
	type	partial GAE security bypass vulnerability
15	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.MethodHandles.Lookup class
	cause	no interception of unreflectGetter() method of java.lang.reflect.MethodHandles.Lookup mirror
	impact	reflective Field access to GAE / system classes
	type	partial GAE security bypass vulnerability
16	origin	com.google.apphosting.runtime.security.shared.intercept.java.lang.reflect.MethodHandles.Lookup class
	cause	no interception of unreflectSetter() method of java.lang.reflect.MethodHandles.Lookup mirror
	impact	reflective Field access to GAE / system classes
	type	partial GAE security bypass vulnerability

17	<b>origin</b>	<code>com.google.apphosting.runtime.security.shared.intercept.java.lang.Class</code> class
	<b>cause</b>	no <code>checkPackageAccess()</code> call in Reflection API methods
	<b>impact</b>	Reflection API calls allowed for prohibited classes ( <code>sun.*</code> package)
	<b>type</b>	partial GAE security bypass vulnerability
18	<b>origin</b>	<code>com.google.apphosting.runtime.security.shared.intercept.java.lang.Class</code> class
	<b>cause</b>	no <code>checkMemberAccess()</code> call in Reflection API methods
	<b>impact</b>	Reflection API calls allowed for declared members
	<b>type</b>	partial GAE security bypass vulnerability
19	<b>origin</b>	<code>com.google.apphosting.runtime.security.shared.intercept.java.lang.Class</code> class
	<b>cause</b>	<code>getProtectionDomain()</code> invocation inside <code>doPrivileged</code> method block
	<b>impact</b>	access to the security sensitive <code>ProtectionDomain</code> object
	<b>type</b>	information leak
20	<b>origin</b>	GAE permissions
	<b>cause</b>	read access allowed for selected JAR files
	<b>impact</b>	reading a code of runtime classes / GAE sandbox itself
	<b>type</b>	information leak
21	<b>origin</b>	GAE deployment
	<b>cause</b>	1+ year old JRE class base (prior to Sep 2013 / JDK7 Update 40)
	<b>impact</b>	unpatched vulnerabilities / exploit vectors
	<b>type</b>	partial GAE security bypass vulnerability
22	<b>origin</b>	<code>org.mozilla.javascript.tools.shell.JavaPolicySecurity\$Loader</code> class
	<b>cause</b>	insecure use of <code>defineClass</code> method of <code>java.lang.ClassLoader</code> class
	<b>impact</b>	arbitrary class loader instantiation (from non-user CL namespace)
	<b>type</b>	partial GAE security bypass vulnerability
23	<b>origin</b>	<code>com.google.common.reflect.Invokable\$MethodInvokable</code> class
	<b>cause</b>	insecure use of <code>invoke()</code> method of <code>java.lang.reflect.Method</code> class
	<b>impact</b>	arbitrary invocation of methods with user provided arguments
	<b>type</b>	partial GAE security bypass vulnerability
24	<b>origin</b>	<code>com.google.apphosting.util.UserClassLoaderHelper</code> class
	<b>cause</b>	insecure use of <code>invoke()</code> method of <code>java.lang.reflect.Method</code> class
	<b>impact</b>	arbitrary invocation of methods with user provided arguments
	<b>type</b>	partial GAE security bypass vulnerability
25	<b>origin</b>	<code>org.apache.commons.beanutils.MethodUtils</code> class
	<b>cause</b>	insecure use of <code>invoke()</code> method of <code>java.lang.reflect.Method</code> class
	<b>impact</b>	arbitrary invocation of methods with user provided arguments
	<b>type</b>	partial GAE security bypass vulnerability
26	<b>origin</b>	<code>org.apache.commons.beanutils.MethodUtils</code> class
	<b>cause</b>	insecure use of <code>invoke()</code> method of <code>java.lang.reflect.Method</code> class
	<b>impact</b>	arbitrary invocation of methods with user provided arguments
	<b>type</b>	partial GAE security bypass vulnerability
27	<b>origin</b>	<code>org.codehaus.jackson.map.introspect.AnnotatedMethod</code> class
	<b>cause</b>	insecure use of <code>invoke()</code> method of <code>java.lang.reflect.Method</code> class
	<b>impact</b>	arbitrary invocation of methods with user provided arguments
	<b>type</b>	partial GAE security bypass vulnerability
28	<b>origin</b>	Class Sweeper
	<b>cause</b>	<code>java.io.zip.ZipFile</code> is a whitelisted / not mirrored class
	<b>impact</b>	execution of user provided code in <code>finalizer()</code>
	<b>type</b>	partial GAE security bypass vulnerability
29	<b>origin</b>	Class Sweeper
	<b>cause</b>	<code>java.nio.ByteBuffer</code> is a whitelisted / not mirrored class

	<b>impact</b>	invocation of prohibited <code>System.gc()</code> call
	<b>type</b>	partial GAE security bypass vulnerability
30	<b>origin</b>	<code>com.google.apphosting.api.ReflectionUtils</code> class
	<b>cause</b>	insecure use of <code>allocateInstance</code> method of <code>sun.misc.Unsafe</code> class
	<b>impact</b>	arbitrary instantiation of prohibited classes ( <code>sun.*</code> package)
	<b>type</b>	partial GAE security bypass vulnerability
31	<b>origin</b>	Class Sweeper
	<b>cause</b>	no inspection of the <code>EnclosingMethod</code> attributes of a Java Class' Constant Pool entries
	<b>impact</b>	access to security sensitive <code>defineClass</code> method of <code>java.lang.ClassLoader</code> class
	<b>type</b>	partial GAE security bypass vulnerability

## APPENDIX B

### URLFETCH RPC SERVICE (PROTOBUF)

```

name: "apphosting/api/urlfetch_service.proto"
package: "apphosting"
message_type {
  name: "URLFetchServiceError"
  enum_type {
    name: "ErrorCode"
    value {
      name: "OK"
      number: 0
    }
    value {
      name: "INVALID_URL"
      number: 1
    }
    value {
      name: "FETCH_ERROR"
      number: 2
    }
    value {
      name: "UNSPECIFIED_ERROR"
      number: 3
    }
    value {
      name: "RESPONSE_TOO_LARGE"
      number: 4
    }
    value {
      name: "DEADLINE_EXCEEDED"
      number: 5
    }
    value {
      name: "SSL_CERTIFICATE_ERROR"
      number: 6
    }
  }
}

```

```
value {
  name: "DNS_ERROR"
  number: 7
}

value {
  name: "CLOSED"
  number: 8
}

value {
  name: "INTERNAL_TRANSIENT_ERROR"
  number: 9
}

value {
  name: "TOO_MANY_REDIRECTS"
  number: 10
}

value {
  name: "MALFORMED_REPLY"
  number: 11
}

value {
  name: "CONNECTION_ERROR"
  number: 12
}
}

message_type {
  name: "URLFetchRequest"

  field {
    name: "Method"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_ENUM
    type_name: ".apphosting.URLFetchRequest.RequestMethod"
  }

  field {
    name: "Url"
    number: 2
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }

  field {
    name: "header"
    number: 3
    label: LABEL_REPEATED
    type: TYPE_GROUP
    type_name: ".apphosting.URLFetchRequest.Header"
  }

  field {
    name: "Payload"
    number: 6
  }
}
```

```
    label: LABEL_OPTIONAL
    type: TYPE_BYTES
    options {
      ctype: CORD
    }
  }

field {
  name: "FollowRedirects"
  number: 7
  label: LABEL_OPTIONAL
  type: TYPE_BOOL
  default_value: "true"
}

field {
  name: "Deadline"
  number: 8
  label: LABEL_OPTIONAL
  type: TYPE_DOUBLE
}

field {
  name: "MustValidateServerCertificate"
  number: 9
  label: LABEL_OPTIONAL
  type: TYPE_BOOL
  default_value: "true"
}

nested_type {
  name: "Header"
  field {
    name: "Key"
    number: 4
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }

  field {
    name: "Value"
    number: 5
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
}

enum_type {
  name: "RequestMethod"

  value {
    name: "GET"
    number: 1
  }

  value {
    name: "POST"
    number: 2
  }
}
```

```
value {
  name: "HEAD"
  number: 3
}

value {
  name: "PUT"
  number: 4
}

value {
  name: "DELETE"
  number: 5
}

value {
  name: "PATCH"
  number: 6
}
}
}

message_type {
  name: "URLFetchResponse"

  field {
    name: "Content"
    number: 1
    label: LABEL_OPTIONAL
    type: TYPE_BYTES
  }

  field {
    name: "StatusCode"
    number: 2
    label: LABEL_REQUIRED
    type: TYPE_INT32
  }

  field {
    name: "header"
    number: 3
    label: LABEL_REPEATED
    type: TYPE_GROUP
    type_name: ".apphosting.URLFetchResponse.Header"
  }

  field {
    name: "ContentWasTruncated"
    number: 6
    label: LABEL_OPTIONAL
    type: TYPE_BOOL
    default_value: "false"
  }

  field {
    name: "ExternalBytesSent"
    number: 7
    label: LABEL_OPTIONAL
  }
}
```

```
    type: TYPE_INT64
  }

  field {
    name: "ExternalBytesReceived"
    number: 8
    label: LABEL_OPTIONAL
    type: TYPE_INT64
  }

  field {
    name: "FinalUrl"
    number: 9
    label: LABEL_OPTIONAL
    type: TYPE_STRING
  }

  field {
    name: "ApiCpuMilliseconds"
    number: 10
    label: LABEL_OPTIONAL
    type: TYPE_INT64
    default_value: "0"
  }

  field {
    name: "ApiBytesSent"
    number: 11
    label: LABEL_OPTIONAL
    type: TYPE_INT64
    default_value: "0"
  }

  field {
    name: "ApiBytesReceived"
    number: 12
    label: LABEL_OPTIONAL
    type: TYPE_INT64
    default_value: "0"
  }

  nested_type {
    name: "Header"

    field {
      name: "Key"
      number: 4
      label: LABEL_REQUIRED
      type: TYPE_STRING
    }

    field {
      name: "Value"
      number: 5
      label: LABEL_REQUIRED
      type: TYPE_STRING
    }
  }
}
```

```
service {
  name: "URLFetchService"

  method {
    name: "Fetch"
    input_type: ".apphosting.URLFetchRequest"
    output_type: ".apphosting.URLFetchResponse"
    options {
    }
  }
}

options {
  java_package: "com.google.appengine.api.urlfetch"
  cc_api_version: 2
  py_api_version: 1
  java_api_version: 2
  java_outer_classname: "URLFetchServicePb"
  java_generic_services: true
}
```

---

## About Security Explorations

Security Explorations (<http://www.security-explorations.com>) is a security start-up company from Poland, providing various services in the area of security and vulnerability research. The company came to life in a result of a true passion of its founder for breaking security of things and analyzing software for security defects. Adam Gowdiak is the company's founder and its CEO. Adam is an experienced Java Virtual Machine hacker, with over 50 security issues uncovered in the Java technology over the recent years. He is also the hacking contest co-winner and the man who has put Microsoft Windows to its knees (vide MS03-026). He was also the first one to present successful and widespread attack against mobile Java platform in 2004.