6.858 Final Project Writeup

PySecure - The Library That Makes Privilege Separation Easy

Sam Fingeret, Qian Long, Kerry Xing

Privilege separation is an essential aspect of security. Privilege separation allows the OS to enforce the policy that certain resources can only be accessed by certain services. Privilege separation also allows for isolation of the different services of an application so that vulnerabilities in one service will not compromise another. As a result, the potential damage that can be done by bugs or attackers can be quarantined. If one service is compromised, damage to other services can be limited or avoided.

Generally, privilege separation is a hard to do correctly due to all of the bookkeeping involved with keeping track of service UID/GIDs. In particular, it is easy to misconfigure permissions when working with UIDs/GIDs. PySecure aims to make privilege separation easy by abstracting away the low level UNIX details such as UNIX permissions, and providing a simple API for creating services with RPCs. This framework will ultimately decrease development time and improve security of web applications.

Implementation (pysecure library in zoobar/)

https://bitbucket.org/kxing/6.858-final-project

Features

Simple RPCs

The developer does not need to worry about creating clients, services, and sockets for RPC functions. They simply add a decorator to functions they want to run as RPCs, and insert some code to make PySecure start an RPC server for the function(s). This interface makes it really easy to modify existing code to add RPCs, since no other code needs to be modified. For example, to make the *authsvc* service an RPC server of a protected function *protected_function*, the resulting code would look like this:

```
@rpc(run_as='authsvc')
def protected_function():
    # Function details

if __name__ == '__main__':
    pysecure_rpc.start_service('authsvc')
```

UID/GID abstractions

The developer does not need to keep track of the UID and GIDs of each service, which is a

tedious task that is prone to error. The developer simply specifies the resource dependencies using PySecure primitives. For example,

```
cred_db_resource = PySecureResource('/zoobar/db/cred/cred.db', 'rw',
'', '')
```

means that *cred_db_resource* is a database file with read/write permissions for the owner, and no privileges for the bundle users or for others. The developer can also specify shared resources by declaring shared bundles. For example,

```
zoobar_bundle = PySecureSharedBundle([list of PySecureResources])
```

would create a bundle with the given resources. Finally, the developer can specify services to be run. For example,

```
authsvc = PySecureService('authsvc', '/', '/zoobar/auth.py',
[cred_db_resource], zoobar_bundle)
```

would create a service named *authsvc*, that is run with the command `/zoobar/auth.py` in the root (chrooted jail) directory, and owns the *cred_db_resource*, and has permissions to shared *zoobar_bundle*. When run, PySecure will examine all of the dependencies, and assign UIDs and GIDs accordingly.

We chose this design because we wanted to have a centralized configuration file that allows developers to see, at a glance, all of the services and resource permissions for their application, similar to Android's use of Manifest files. This design is definitely an improvement over Zoobar's original configuration setup, which had security logic scattered in the `chroot-setup.sh` and `zook.conf` files.

Sandboxing

It is difficult for the current framework to support sandboxing of user code, since this requires both a dynamic chroot after initialization as well as the creation of a new RPC API server to interface with the sandboxed code. As a proof of concept that it is possible for a sandbox API to be implemented with the PySecure framework so that developers do not have to work with UIDs or chroot. We have provided a method for doing so in the *sandbox* branch using functions in `pysecure_sandbox.py`. This module provides a `run_sandboxed_code(pcode, userdir, api)` function which takes as arguments the user code (to be sandboxed), the directory for the code to be jailed, and an API dictionary containing a list of functions for the sandboxed code, and returns the output of the sandboxed code.

Currently, the UID for the sandboxed code process and the API UID are hardcoded in this module, but it would not be difficult to have this module interface with the PySecure configuration

file to allow the developer to specify API resources. In addition, it would not be too difficult to allocate a UID to the sandboxing module from the PySecure UID allocation code.

The sandbox API in the *sandbox* branch is found here: https://bitbucket.org/kxing/6.858-final-project/src/75f9c748b3eba040e3c05a8a5c6a035a9fcb1131/zoobar/pysecure/pysecure_sandbox.py?at=sandbox

Easy jail setup
In addition to the privilege separation API, we have included a set of scripts that lets the developer easily set up the jail environment by specifying a list of paths to copy to the jail (ie. `jail_resources.txt`). Our setup script will take care of creating the directory structure and copying dependencies for binaries. This can also be extended for more custom setups.

For resource initialization (such as creating database files in the case of zoobar), the developer will have to define their own script (ie. `pysecure_initialize_resources.sh`), which will be run at jail creation. This gives developer more flexibility for initializing their resources because it is hard to anticipate the different types of resources developers will be creating.

Deployability

We ported the Zoobar application to Apache. Setting up Zoobar was simple and relatively painless. Indeed, configuring Apache to do chrooting correctly was much harder than getting Zoobar to work under Apache.

The jail setup code for Apache can be found in the *apache* branch of our BitBucket repo: https://bitbucket.org/kxing/6.858-final-project/src/83fca51d16240b0640cbeda76f3fd112c8913973/?at=apache Instructions for configuring Apache to run Zoobar can be found in the README file.

PySecure API Documentation

package pysecure_rpc

@rpc(run_as=None)
Decorator for an RPC function. The *run_as* parameter represents the service that executes the RPC call.

start_service(service_name)
Starts an RPC server with the given *service_name*.

package pysecure

PySecureResource

PySecureResource(path, owner_permissions, bundle_permissions, global_permissions, owned_by_dynamic_svc=False)

Creates a PySecureResource object representing the resource at the specified *path*. *owner_permissions*, *bundle_permissions*, and *global_permissions* are strings optionally containing ‘r’ (read permissions), ‘w’ (write permissions), and ‘x’ (execute permissions). If your dynamic service is the owner of the resource, set *owned_by_dynamic_svc* to True.

PySecureBundle

PySecureBundle(resources, used_by_dynamic_svc=False)

Creates a PySecureBundle object representing a shared bundle consisting of the given resources. *resources* is an iterable of PySecureResource’s. If your dynamic service needs to use the bundle, you should set *used_by_dynamic_svc* to True.

PySecureService

PySecureService(service_name, directory, command, owner_resources=[], bundle=None, runs_with_root=False)

Creates a PySecureService object representing a service named *service_name*. The service will be started by running *command* in *directory*. The service has full ownership of the resources in *owner_resources*, and shares the resources in the *bundle*. The service is run as a user process by default. If the service needs root privileges, then *runs_with_root* should be set to True.

PySecureConfig

PySecureConfig(self, app_name, uid_range, gid_range, dynamic_svc_uid=None, dynamic_svc_gid=None)

Creates a PySecureConfig object representing a configuration setting for the app with name *app_name*. PySecure will allocate UIDs in the range *uid_range*, which is a tuple of two numbers representing the lowest and highest UIDs that can be allocated by PySecure. Similarly, PySecure will allocate GIDs in the range *gid_range*, which is a tuple of two numbers representing the lowest and highest UIDs that can be allocated by PySecure. If your dynamic service needs to own one of the listed resources, you should set *dynamic_svc_uid*. If your dynamic service needs to own one of the shared bundles, you should set *dynamic_svc_gid*.

config.register_resources(resources)
Registers a list of resources.

config.register_bundles(self, bundles)
Registers a list of bundles.

config.register_services(self, services)
Registers a list of services.

PySecure

PySecure.run_with_config(config)
Starts up the application services specified by the PySecureConfig object *config*. The UIDs and GIDs are set accordingly.

Jail Setup Scripts

jail_resources.txt
Developer puts paths of folders and binaries they want in the jail environment

pysecure_initialize_resources.sh
Defined by developer to initialize resources.

pysecure_create_jail.sh
Run this to create jail environment. Copies folders and binaries specified in jail_resources.txt over to the jail. Runs pysecure_initialize_resources.sh.