PSPY: A Privilege Separation Framework for Python Applications

Ge (Jackie) Chen, Santiago Perez De Rosso, David Way

6.858 Final Project Report

Abiding to the principle of least privilege is cumbersome. The developer must think about running different parts of the program under separate processes with reduced privileges, setting file permissions for restricted access, and creating a remote procedure call system for internal application interactions. In order to simplify and automate this work, we have developed pspy, a framework for privilege-separating Python apps. To use our framework, developers simply need to organize their application into “services” and run a bootstrapping script for setup. The framework also comes with a static analysis tool that developers can run after bootstrapping to see the new state of their application, in terms of services, IDs, and file permissions. We evaluated our framework with three examples: the Zoobar web application, the Flask Minitwit app, and a Django paste-bin app. Our framework is simple (the developer only has to write very few lines of code to privilege-separate his app) yet powerful (it can address non-trivial privilege separation needs).

Design

We wanted to design the framework API to be as simple as possible for the developer. The regular approach to privilege-separation (without pspy) involves: (i) creating a socket for each service, (ii) modifying all "client" calls so that they just write to this socket using an RPC library, (iii) setting up a jail directory to which the services will be jailed and (iv) setting the appropriate permissions for the files. This task is cumbersome. Moreover, once you have created this setup, it is very hard to modify. If you want to change it, for example to put two services into one process, you would have to modify the startup script, change uids/gids appropriately and so on.

With pspy, the user merely: (i) annotates the classes/functions with a "service" decorator and (ii) runs our bootstrap script. The "service" annotation has several optional arguments that let the developer customize how that service should work: the "perms" keyword argument allows the developer to specify a dictionary of filepaths to permissions, the "group" keyword to specify a group for a service, and the "additional groups" to specify additional groups. Moreover, services can be grouped together into a "service provider" so different classes or function can run under one process. The developer does not have to worry about the underlying mechanisms for privilege separation--pspy will handle socket creation, process launching, the appropriate setting of uids/gids and so on. If the developer wants to make a change to put two services into one process, he merely has to group them into one service provider and pspy will figure out the rest.

Implementation

With the decorator, we are able to parse the application's code to detect services as well as easily modify Python classes and functions to create the RPC server and client sides for each service. We provide two scripts tools that the user can run: bootstrap and analyze.

Bootstrap script

When running bootstrap, the user can specify a directory to which the application should be jailed to, which defaults to '/jail'. All of the Python application files and extra user-specified files are copied to this directory, as well as 'sys.path'. All services are jailed to this directory using chdir/chroot. The user can also specify a script file--to be run after the application is copied to the jail--for additional actions such as creating database files.

Bootstrap parses all of the Python application files (using Python's ast.NodeVisitor class). Each class and function definition that is decorated with '@pspy.service(...)' is added to a stored dictionary with the service provider as the key and the decorator keyword arguments as the value. If a class or function is decorated with the same service provider name, those services will run under one uid process, and the values in the stored dictionary with that service provider key will be combined. This dictionary is used to choose ids based on the service and group names, set file permissions, and launch the RPC server processes for each service provider.

Analyzer script

For large applications, it can be difficult for developers to manage many services and file permissions when privilege-separating an application. To help the developer take a snapshot of their work and easily check if services have various permissions over various files, we wrote a static analysis tool that outputs the current state of the privilege-separated application.

First, the user must run the bootstrap script to copy all of the application files to the proper jail directory. Then, the user is free to run the pspy-analyze script. The script needs to receive the same jail directory, app files, uid, and gid arguments as bootstrap such that it can collect the proper UID's and GID's for each service as it parses through the application. Given these values, the analyze tool then searches the jail building lists of files and directories and their corresponding permissions for each service. Once finished, the script displays the information it has found in a usable manner for the developer to take stock of each service's power.

Evaluation

Zoobar

Naturally, we wanted to apply our framework to an application that has non-trivial privilege-separation needs. Given a slightly improved fresh zoobar lab2 branch, we added decorations specifying one service for each method concerning the bank database and added decorations specifying one service for methods concerning the cred database. (See Fig 1 for examples.)

```
@pspy.service(group="BankGroup", perms={"zoobar/db/bank/": 0700}, service_provider="Bank")
def transfer(sender, recipient, zoobars, token):
    bankdb = bank_setup()
    senderp = bankdb.query(Bank).get(sender)
    recipientp = bankdb.query(Bank).get(recipient)
```

Fig 1(a) - a snippet of Bank's transfer service.

```
@pspy.service(service_provider="Bank")
def balance(username):
    db = bank_setup()
    person = db.query(Bank).get(username)
    return person.zoobars
```

Fig 1(b) - a snippet of Bank's balance service.

```
@pspy.service(group="AuthGroup", perms={'zoobar/db/cred/': 0700}, service_provider="Auth")
def login(username, password):
    db = cred_setup()
    person = db.query(Cred).get(username)
    if not person:
```

Fig 1(c) - a snippet of Auth's login service.

With the proper permissions for each service's respective database directory, the check_lab2.py script was able to pass exercises 7 and 9 (the bank and auth separation checks). In order to help with creating the many zoobar dependencies inside of /jail, we made use of the --script flag of the bootstrap script (that allows the developer to specify a script that should run after the jail has been set) to set up app dependencies and database initialization.

Minitwit

We applied our framework to a Flask microblogging application. Since this application had one database file with three tables, we could not split the application into many services (without splitting the database). Nonetheless, we create one service that handled database I/O. The RPC servers and client startup run correctly for this example, but unfortunately, our system uses JSON for encoding parameters across RPC calls and was not able to encode certain objects Minitwit used as parameters (the sqlite3 connection object). In order to fix this, the application must ensure that JSON compatible parameters are used, or JSON (or our framework) must provide a wrapper class that allows such encoding. We found that this technicality was unrelated to our project goals, and decided to spend time on more pressing issues.

Paste-bin

We decided to try our framework with a Django app as well. To do this, we took dpaste, a popular paste-bin app written with Django, and added an "evaluate" service that will evaluate Python code. The service is a very trivial one, but our goal with this example was to serve as a proof of concept that it is possible to use pspy in Django apps as well. But Django is a very different framework from Flask as it is more "heavy-weight", providing several constructs that automate the process of building a web application. Although it is possible to use pspy in a Django app, pspy could be extended so as to provide Django-like constructs, to give a better experience to developers using Django.

Conclusion

We developed pspy, a framework for privilege-separating Python applications. We believe our framework is simple to use (the changes in the developer's codebase only involve annotating functions/classes with the service decorator) yet powerful (it is possible to specify groups, additional groups, permissions of files and group services together). We evaluated our framework with three examples, two Flask apps (Zoobar and Minitwit) and a Django app (a paste-bin) app that show that our framework can be useful in practice.