User Impersonation

1. Overview

UNIX is lacking a fundamental feature that allows for temporary user switches at run time. The `setuid()` group of system calls allow for a permanent switch but offer no return path. My goal during this project is to rectify this by creating a user impersonation sub-system into FreeBSD using dynamically loadable kernel modules.

Since the motivation for building this is to provide a fast and secure method for PHP to run without `safe_mode` restrictions, this document will make a lot of references to The Apache Web Server and some of its modules. This document will include a fully functional implementation based on Apache.

2. Implementation

To solve this I've taken the steps necessary to build a kernel module (FreeBSD kld) that provides impersonation services, allowing any user to become any other user and return again. There are three basic pieces to this: the first being the loadable kernel module (impersonate.ko), the second is the userland library that wraps the impersonate system calls in a nice simplified interface, and the third is a userland tool called `impadm` that is similar in syntax to FreeBSD's `ipfw`, allowing admins to build the impersonation rule-set.

2.1. Kernel Module (impersonate.ko)

The kernel module is responsible for providing the impersonation facility, defining the rules, and maintaining session state.

Rules are defined very simply using an order number, requesting user id (-1 = all), impersonating user id (-1 = all), and an action or rule (allow or deny). The order of the rules is based on the rule number and each rule is visited in this order. The first rule that matches a request is assumed to be the most current acurate rule and will be used to determine whether or not an impersonation request is allowed.

Sessions are an essential piece of the impersonation system. This is what allows a process to return to its original state without a specific rule defined. Sessions remain active indefinitely (or until the process is dead) while the process is in an impersonated state but upon returning a time to live (TTL) counter begins ticking down and will ultimately result in the session being deleted. A new session is created if one doesn't already exist.

2.2. Userland Library (libimpersonate.a|so)

The impersonation library exposes the kernel module to userland applications. This wraps the complexity in making direct system calls in a simple interface (just a few functions).

FIXME (sdeming):

TODO: sample usage...

Also planned is a simple C++ class library that will provide some basic Command object decorators allowing an easy transition for existing OO frameworks that utilize the Command design pattern.

2.3. Userland Admin Tool (impadm)

The default rule-set is to deny impersonatation to everyone. This is achieved by a very simple fact: If a user is requesting impersonation and no rule has been defined for this user then access is denied. Quite simple really, and it gives us piece of mind when building our rule-sets

Building your rule-set will depend on your specific needs but can be easily summarized as adding allow/deny rules with specific ordering. In an Apache installation that hosts several customers in a virtual shared hosting environment you will have a very simple rule-set that can be built using impadm in the following way:

```
# impadm add 1000 allow apache all
# impadm add 1 deny all root
```

You can see the current active rule-set by invoking:

```
# impadm show rules
1       -1     => 0     deny (hits: 0)
1000    500    => -1    allow (hits: 0)
```

Notice that the uid's are presented here instead of the usernames. This output is generated by the kernel module and will eventually be coded to pass the details straight to the caller for formatting and display. For now this works well and anything additional will have to come after all security concerns are addressed.

Reading the output of the "show rules" command you can see the following columns of information:

1. rule number
2. requesting uid

3. activating uid
4. action (deny or allow)
5. a hits counter

For the time being a uid of -1 means all users. Since uid_t is generally an unsigned number this should actully evaluate to the highest allowed uid in the system. Hopefully it isn't being used, if so then there is a potential vulnerability and it is noted as a serious security issue.

Rules are always presented in the order they are visited. The hit counter gives you a running tally of how many times an impersonation request was allowed or denied because of a specific rule.

Deleting rules from the rule-set is also very simple:

```
# impadm del 1000
```

Will delete rule number 1000, so any further requests by apache to impersonate anyone will be denied.

To view current sessions we have the "show sessions" command that would produce results like the following:

```
# impadm show sessions
7779   500  /500   => 500  /500   : ttl 60       (hits: 6       )
7782   500  /500   => 1003 /1003  : ttl 60       (hits: 5       )
7781   500  /500   => 500  /500   : ttl 60       (hits: 4       )
7780   500  /500   => 1004 /1004  : ttl 60       (hits: 15      )
7728   500  /500   => 1003 /1003  : ttl 60       (hits: 7       )
7729   500  /500   => 1001 /1001  : ttl 60       (hits: 5       )
6512   500  /500   => 1001 /1001  : ttl 60       (hits: 5       )
7717   500  /500   => 500  /500   : ttl 59       (hits: 4       )
6519   500  /500   => 1002 /1002  : ttl 60       (hits: 9       )
6510   500  /500   => 1003 /1003  : ttl 60       (hits: 7       )
6509   500  /500   => 500  /500   : ttl 59       (hits: 6       )
6508   500  /500   => 1003 /1003  : ttl 60       (hits: 7       )
6516   500  /500   => 1002 /1002  : ttl 60       (hits: 5       )
6511   500  /500   => 500  /500   : ttl 60       (hits: 12      )
```

In the above output the following columns are present:

1. pid
2. requestor uid / gid
3. impersonating uid / gid
4. ttl (time to live)
5. a hit counter

Once again we have a statistical hit counter that tells us how many times this process has made an impersonation request, it will be lost when the session is deleted. TTL begins to tick down once a process returns to its normal state (when requestor and impersonator uid/gid's are the same), when it hits 0 the session is deleted.

3. Case: The Apache Web Server

UNIX is lacking severely one feature that would make virtual web hosting significantly more secure and less of a bother for end users. Consider the following facts:

- Apache runs as a single user with read access to all files it serves. Actually Apache excersizes privilege seperatation in that it starts as user root (required to bind port 80) and then switches to the more restricted user defined in httpd.conf as User/Group (usually nobody, apache, or httpd).
- mod_perl cannot be made secure in a shared hosting environment.
- mod_php has safe_mode, but that's more of a bandaid than a solution.
- The apache user requires read access to all files it serves, meaning user files must be globally readable by at least one user.
- suexec helps to solve many of these issues but incurs a performance penalty (double fork/exec).

3.1. Without Impersonation

The following is an illustration that shows from a high level, how apache processes a given request for a page, without user impersonation:

Apache Without Impersonation

To put it into words, Apache receives the request from a browser asking for a certain type of document. The document handler then reads and executes the file through whatever means defined and delivers the resulting output to the browser. This entire activity takes place in a single context, as the user that owns the Apache process. The one exception to this rule is CGI when suexec is enabled, which kindly forks a new process then switches to the real user before invoking the script. This would be ideal except for the cost of a fork/exec for each request. On a busy web server this can lead to some performance issues.

3.2. With Impersonation

This illustration shows how Apache processes a request when user impersonation is active:

Apache With Impersonation

As you can see, we're introducing a new security layer by temporarily becoming the VHost user prior to processing the request. The generic Apache user is no longer required to have read access to the files it is serving, keeping prying eyes away. Further, Apache modules such as mod_perl and mod_php can be invoked as the proper user now allowing for a level of security that was previously unavailable. With user impersonation, PHP safe_mode is no longer a necessity.

Also worth noting is that suexec is no longer in the picture. This is more of a nice side-effect than anything, but by itself makes the project worth while. Without the need of a fork/exec the inherent performance penalty of securely running CGI scripts is reduced by up to 50%.

3.3. Setting Up For Testing

Our first job before we begin work should be to set up a test environment and identifying the behavior of an *unimpersonated* system.

Apache is installed and has the following VirtualHost defined:

```
<VirtualHost 192.168.0.50>
  ServerName     phptest1
  DocumentRoot   /usr/home/phptest1/htdocs
  User           phptest1
  Group          phptest1
  AddType        application/x-httpd-php .php
  AddHandler     cgi-script .cgi
  ScriptAlias    /cgi-bin "/usr/home/phptest1/cgi-bin"

  <Directory /usr/home/phptest1/htdocs>
    Options Indexes Includes ExecCGI FollowSymLinks
    AllowOverride All
  </Directory>
</VirtualHost>
```

We also have a very simple script called `test.php` in `/usr/home/phptest1/htdocs` that contains the following:

```
<?php
  system('who am i');
?>
```

So with impersonation turned off, we would expect to see the following output in our web browser when we request our test.php page:

```
nobody           tty??    Sep 22 21:47
```

This indicates that Apache is run as user `nobody` and the PHP script request is being processed as the base Apache user. So our initial test will indicate that everything is working just like any typical web server.

At this point, our only test case is to prove that an impersonated request is in fact processed as the user that actually owns this VirtualHost (phptest1). Our initial test is a failure, which is exactly what we expect.

3.4. Applying Impersonation

The easiest place to implement the impersonation routines is in the Apache source module `http_request.c`, wrapping the `process_request()` function. By entering a state of impersonation before invoking `process_request()` we are essentially forcing this on all requests, probably not the best idea for a production installation but it is perfect for testing and proving the concept.

Now that we have Apache patched and wrapped up with the impersonation code we can build it, run it, and try out our test. What we expect to see is the user `phptest1` in our call out to `who am i`. This is the actual result from this test:

```
phptest1         tty??    Sep 22 22:20
```

The test is a success.

3.5. Summary

Impersonation can be effectively used to allow for safe script processing as the correct user without the cost overhead associated with a double fork/exec pair for each request as is the case with your typical CGI script running through `suexec`

Further tests acknowledge that Apache, when patched as above, no longer requires read or execute access to the files it processes. Since the impersonation occurs before checking up on or opening the file, a very important security concern is alleviated. More specifically the potential to leak critical information to wandering users utilizing the same server is gone.

Further testing also revealed some critical flaws with the forced impersonation for every request. For example, in one Apache module called `mod_throttle` shared memory is utilized and must be available. Since shared memory is protected, once the impersonation routine kicks the Apache process is no longer allowed to access any of the shared memory segments it created.

In the end, the idea is proven to be valid but the implementation has to be done with great care.

4. Concerns

4.1. Security

- Potential showstopper: If impersonated user can control code execution dynamically then in theory they can force the process to return to its original user while retaining control. This in turn would allow them to impersonate any other impersonable user.
- A race condition exists because PID is used as a key to identify sessions. Obviously a production ready implementation of this idea would have to be done with much more care.

4.2. Performance

Implemented properly, there should be no noticeable performance reduction. In the case of Apache, every site request will cause two system calls to be invoked.

4.3. Administration

Using the `impadm` tool to administer the rules for impersonation is fairly straight forward. The real concern is in making sure the defined rules are clean from potential misuse.

4.4. Intrusive

A kernel module introduces a certain complexity that could be a very big issue when performing OS upgrades. If the module is not maintained with each advancement of the OS there will be no safe upgrade path available.

5. Conclusion

With user impersonation enabled we can run Apache in a much more secure environment. Gone are the days that Apache has to have read permissions on all of the files it serves providing a sort of back-door for users to access other users files. With this solution Apache will be effectively running as the actual user for the duration of the request.

There are many other applications besides Apache that could benefit from this technology.

6. Personal Notes

While the examples provided here is a real world working implementation many fundamental issues still exist that prevent me from pushing this into a production setting. The idea that a process can be manipulated by the impersonated user is probably a show stopper and will not be easily resolved. And while that issue in particular was a concern from the start, I still wanted to make this work. The lesson in FreeBSD and UNIX internals was well worth the effort.