STEVEN ALEXANDER

improving security with homebrew system modifications

Steven is a programmer at Merced College. He has been using FreeBSD since version 2.2.6 and still loves it.

alexander.s@mccd.edu

IN THIS ARTICLE I DISCUSS THREE modifications I've developed for FreeBSD. The first modification is a variant of the MD5-crypt mechanism, which uses an increased number of iterations in the internal loop of the crypt function. It also hashes in a constant string during the first iteration of the core loop. Increasing the number of iterations causes password hashing to require more computation time. This should not significantly impact most systems because they don't spend much of their time authenticating users. An attacker, on the other hand, wants to be able to guess millions of possible passwords per second. The attacker's efforts can be severely impacted. The constant string helps to prevent the use of standard password-cracking tools.

Another modification that I've developed is for the gcc compiler (version 2.9.5) as distributed with FreeBSD 4.8–4.10. This modification should work on other operating systems. The change I've made adds two new compile-time options to gcc. One option randomly adds up to 1 megabyte to the stack size of the function main(). The other option adds up to 16KB to the stack size of all functions. The first option is enabled by default. The second option is disabled by default and should be used very sparingly as it can have severe consequences in the way of wasted memory. Changing the stack layout of a program can defeat many buffer overflow exploits. This technique was introduced by researchers at the University of New Mexico.[1] An attacker who can tailor tools for your system will be able to defeat this defense.

More advanced randomization techniques than those used by Forrest et al. have been developed.[2] Run-time randomization of a program's memory layout is stronger but can have negative performance consequences. Load-time stack randomization is also stronger, since it is dynamic, and is currently available in RedHat Linux, OpenBSD, and PaX. I've included a patch, below, to add load-time stack randomization to FreeBSD.

For more information on buffer overflows and protection mechanisms, see the SmashGuard buffer overflow page at Purdue University.[3] More advanced memory randomization and protection measures are available with PaX and OpenBSD.[4]

The crypt modification provides hard security in that the increase in difficulty for an attacker to mount an

offline password guessing attack is absolute as long as no serious cryptanalytic breakthrough is achieved against MD5-crypt. It also provides a soft measure of security through obfuscation. The modified crypt mechanism will be incompatible with other systems and standard password-cracking tools. A knowledgeable attacker can modify his or her tools to suit your system and try an offline attack, even though, because of the increased computation time, it will be less likely to succeed. Attackers who do not understand the need to modify their tools or are unable to do so will have no chance of success.

The security provided by load-time stack randomization is much more solid than that provided by compile-time randomization. In the latter case, an exploit simply needs to be tailored to a given system to work on that system. The reason it is useful is that many attackers don't have the skills or access to a particular target system needed to tailor an exploit to that system. On the other hand, load-time randomization introduces the property that many exploits will not work except by brute force, even if the attacker has access to the compiled program.

Most attackers are not expert programmers or security gurus, but tend to be kids or disgruntled employees (current or former). These attackers depend on tools that are developed by more skilled programmers. The programmers who write these tools make assumptions about the conditions of the target system. If these conditions do not hold, the tools will fail. This enables us to use both diversity and randomization for positive gain.

Randomizing Stack Sizes (Compile-Time)

In order to make gcc add a small random buffer to the top of a function's stack, I read in a 32-bit number from `arc4random()` and mask it to get the size I want. I then modify the frame offset in `init_function_start` by that many bytes and use two new compiler flags that control whether to modify `main()` and/or all other functions. The flags can be invoked using `-f[no-]randomize-stack-main` and `-f[no-]randomize-stack-all`. The latter is disabled by default. Several machines have been rebuilt with this patch in place and have been running without any problems for several months. I'm also running ProPolice/SSP on some of these machines and have not had any problems.[5] Use ProPolice (or StackGuard)—you'll be happier for it. If I've done anything taboo, I hope some of the gcc experts out there will clue me in. The gcc source lives in /usr/contrib/gcc .

function.c

In function.c, `init_function_start` requires modification. Changes are shown in bold.

```
  . . .
void
init_function_start (subr, filename, line)
       tree subr;
       char *filename;
       int line;
{
int random_dword = 0;
  . . .
/* We haven't had a need to make a save area for ap yet.
*/
arg_pointer_save_area = 0;
/* No stack slots allocated yet. */
```

```
  if(flag_randomize_stack_all ||
(flag_randomize_stack_main && \
      (strcmp(current_function_name,"main")==0) ) )
  {
       random_dword = arc4random();
       if(strcmp(current_function_name,"main")==0)
            random_dword = random_dword & 0x000ffffc;
       else
            random_dword = random_dword & 0x00003ffc;
#ifdef FRAME_GROWS_DOWNWARD
       frame_offset = -random_dword;
#else
       frame_offset=random_dword;
#endif
     }
     else          /* no randomization */
     {
          frame_offset = 0;   /* Original code. If you
                 keep an extra frame_offset = 0,
                 the code won't work. */
     }
```

flags.h

The following entries must be added to the end of flags.h:

```
...
/* Nonzero means use stack randomization for main() */
extern int flag_randomize_stack_main;
/* Nonzero means use stack randomization for all functions */
extern int flag_randomize_stack_all;
```

toplev.c

These flags are then defined and set in toplev.c:

```
   . . .
  int flag_no_ident = 0;

  /* Nonzero means randomly increase the stack space used
by main by up to 1 megabyte */
  int flag_randomize_stack_main = 1;
  /* Nonzero means randomly increase the stack space used
by all functions */
  int flag_randomize_stack_all = 0;

   . . .
  {"ident", &flag_no_ident, 0,
     "Process #ident directives"}  ,
  {"randomize-stack-main", &flag_randomize_stack_main, 1,
     "Enable stack randomization for main" },
  {"randomize-stack-all", &flag_randomize_stack_all, 1,
     "Enable stack randomization for all functions" }
};
```

To rebuild gcc you should:

1. `cd /usr/src/gnu/usr.bin/cc`
2. `make obj`
3. `make depend`
4. `make all install`

Afterwards, gcc will pad the stack for main() on all newly compiled programs by default. This can be turned off by using the option -fno-randomize-stack-main. Optionally, padding can be used on all functions by specifying -frandomize-stack-all. This flag can impose a very large overhead, particularly on programs that use recursive functions. Enable it at your own risk.

The entire system can be recompiled using stack padding (for `main()` only) by:

1. `cd /usr/src`
2. `make buildworld`
3. `make buildkernel`
4. `make installkernel`
5. `reboot`
6. `make installworld`
7. `reboot`

Randomizing Stack Sizes (Load-Time)

I have also implemented load-time stack randomization by modifying /usr/src/sys/kern/kern_exec.c. The changes are minor and similar to those used in the gcc patch. In `exec_copyout_strings`, the `vectp` pointer, which becomes our stack base, is modified. This modification has been tested on FreeBSD 4.8–4.10 and 5.2.1.

This feature can be defeated by brute force or possibly in conjunction with a format string attack. In the case of brute force, the process is noisy and the attack can be stymied using techniques such as those in SegvGuard for Linux.[6] Unfortunately, such a tool is not currently available for FreeBSD. Still, load-time randomization is more difficult to defeat than compile-time randomization, as it is dynamic and an attacker must brute-force the stack addresses rather than simply analyzing the binary. Format strings can be used to defeat this technique but only under particular circumstances.

```
 . . .
#include <sys/libkern.h>
 . . .
register_t *
exec_copyout_strings(imgp)
  struct image_params *imgp;
{
 . . .
/* local variables */
 . . .
  int random_offset;
 . . .
  /*
   * The '+ 2' is for the null pointers at the end of each
of the arg and env vector sets
   */
    vectp = (char **)
      (destp - (imgp->argc + imgp->envc + 2) *
sizeof(char*));
  random_offset = arc4random();
  random_offset = random_offset & 0xffffc;
  vectp-=random_offset;
```

The kernel needs to be rebuilt to use the new changes:

1. `cd /usr/src`
2. `make buildkernel`

3. `make installkernel`
4. `reboot`

Creating a New Crypt Mechanism

A few months ago, I was re-reading parts of *Practical UNIX and Internet Security*[7] and noted the authors' suggestion that system administrators modify the crypt routine on their UNIX systems to loop more than the standard 25 times, in order to prevent attackers from using a standard password cracker. I decided to implement this on some of my systems. This modification works on FreeBSD 4.8–4.10 and 5.2.1. The existing source code for 5.2.1 looks slightly different, but the changes are the same.

If I were simply to change the existing source code and recompile, all the accounts on my systems would stop working. That is why FreeBSD allows new crypt mechanisms to be added without replacing the original mechanisms. All new passwords are hashed using the mechanism that is configured in login.conf.

Rather than modify the old DES-based crypt mechanism, I modified a copy of the MD5-based crypt mechanism, which is much stronger. MD5-crypt was designed by Poul-Henning Kamp and uses Ron Rivest's MD5 hash algorithm.[8] I do not suggest arbitrarily modifying the crypt mechanisms unless you have real cryptographic expertise, as you may inadvertently weaken the algorithm. I have only increased the number of iterations of the algorithm and hashed in a constant value; everything else is intact.

Passwords that are hashed on your system using the new crypt mechanism will not be breakable with an unmodified password cracker. To make the job of an attacker more difficult, back up and remove the libcrypt source code from your servers after the new libcrypt has been installed. An attacker can still analyze the binary code to find out how it was modified, but many attackers do not have these skills. If an attacker is unable, or unknowingly neglects, to do this, his or her offline attack will not succeed.

On FreeBSD and many other systems, non-root users are not able to see the stored password hashes. These hashes are still valuable to an attacker. Many attackers copy the password file after a break-in so that the passwords can be tried on related systems to which the attacker does not have access or reused on the same system if the hole the attacker used to break in is patched.

Adding a new crypt mechanism to FreeBSD turns out to be pretty easy. The source for libcrypt is located at /usr/src/lib/libcrypt. Three files need to be changed to create a new mechanism: crypt.c, crypt.h, and Makefile. A file must also be created that contains your new mechanism.

crypt.h

The header file crypt.h contains the prototypes for the different crypt mechanisms. You can name your new mechanism whatever you like; mine looks like this:

```
 . . .
char *crypt_md5_local(const char *pw, const char *salt);
 . . .
```

crypt.c

crypt.c contains a data structure named `crypt_types` that contains the name of the mechanism, the function to call to use the mechanism, and the magic value that is prepended to passwords that use this mechanism. My entry to the list looks like:

```
 . . .
```

```
  {
    "md5local",
    crypt_md5_local,
    "$4$"
  },
  {
    NULL,
    NULL
  }
 . . .
```

crypt-md5-local.c

In order to create a new crypt mechanism, I copied the file crypt-md5.c to crypt-md5-local.c. You must modify the new file slightly. Rename the function `crypt-md5` to `crypt-md5-local`. Near the beginning of the function, the magic value should be changed from `$1$` to `$4$`. The magic string `$3$` is not used in FreeBSD 4.x but is used in FreeBSD 5.x for the NT hash algorithm. Your changes should look like this:

```
 . . .
char *
crypt_md5_local(pw, salt)
  const char *pw;
  const char *salt;
{
  static char*magic = "$4$";   /*
                     * This string is magic for
                     * this algorithm. Having
                     * it this way, we can get
                     * better later on.
                     */
 . . .
```

Further down in the code, a `for` loop iterates 1000 times to form the core of the MD5-crypt mechanism. You can replace the value 1000 with anything you like. Increasing the number significantly will make password cracking very difficult; however, increasing it too much could slow the system unnecessarily. If, for some reason, you need password database information to be interoperable on multiple systems, each system will need to use the same value in its modified crypt mechanism. Here, I have changed the number to `8000`. I also have hashed in the constant string `mercedcollege`; this prevents an attacker from performing an offline password cracking attack without modifying his or her tools to suit. Modify this string to something of your own choosing.

```
 . . .
  for(i=0;i<8000;i++) {
    MD5Init(&ctx1);
  if(i==0)
    MD5Update(&ctx1,"mercedcollege",strlen("mercedcol-
lege"));
    if(i & 1)
      MD5Update(&ctx1,pw,strlen(pw));
 . . .
```

Makefile

The name of the file that contains your new crypt mechanism must be included in Makefile:

```
SRCS=    crypt.c crypt-md5.c crypt-md5-local.c md5c.c
misc.c
```

/etc/login.conf

After these changes are made be sure to `make` and `make install`. /etc/login.conf can be modified to use this new method as the default. The entry should look like:

```
:passwd_format=md5local:\
```

Afterwards, you must run `cap_mkdb /etc/login.conf`.

To install the new libcrypt:

1. `cd /usr/src/lib/libcrypt`
2. `make`
3. `make install`
4. `reboot`

REFERENCES

1. S. Forrest, A. Somayaji, and D. Ackley, "Building Diverse Computer Systems," *Proceedings of the 6th Workshop on Hot Topics in Operating Systems* (1996): http://www.cs.unm.edu/~immsec/publications/hotos-97.pdf.

2. Monica Chew and Dawn Song, "Mitigating Buffer Overflows by Operating System Randomization," Tech Report CMU-CS-02-197 (December 2002); PaX, http://pax.grsecurity.net/.

3. See https://engineering.purdue.edu/ ResearchGroups/SmashGuard .

4. See http://pax.grsecurity.net/, http://www.openbsd.org.

5. Hiroaki Etoh, "ProPolice: GCC Extension for Protecting Applications from Stack-Smashing Attacks," IBM (April 2003): http://www.trl.ibm.com/projects/security/ssp/.

6. Nergal, "The Advanced return-into-lib(c) Exploits: PaX Case Study": http://www.phrack.org/phrack/58/p58-0x04.

7. Simson Garfinkel and Gene Spafford, *Practical UNIX and Internet Security* (Sebastopol, CA: O'Reilly, 1996).

8. Ron Rivest, "The MD5 Message-Digest Algorithm," RFC 1321 (April 1992).