Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Digital Signature Implementation Example in Linux

Author: Kirill Brazhnikov, Software Engineer, Auriga, Inc.

Introduction

One of the most important questions in today's world is the question of confidence in received data. For example, user A sends data D to user B via email. How can user B be sure that the received data is the same data that was sent by user A? One possible way of resolving this issue is using a digital signature (DS). The following requirements apply to a DS:

1) The signature content should depend on the signed message;
2) The sender's unique information should be used in a signature;
3) It should be easy to create a signature;
4) It should be impossible to falsify a signature computationally;
5) A signature should be small.

This article considers a DS implementation example for binary file integrity checking in Linux (64-bit ELF). We will use a direct DS when only a sender and a recipient are communicating (without a third party/an arbitrator). We will need a private encryption key and a public key (certificate) for this. The sender creates both keys. User A signs an executable file and passes the certificate to user B with the help of safe delivery means. After this, user A sends a signed file to user B. User B runs the received file; if the binary file is corrupted, user B will receive a message that DS verification has failed. To implement this solution, we will need a program for signing binary files and a code that verifies DSs.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

DS implementation example

DS implementation includes the following steps:

1. MD5 source binary file generation;
2. The creation of two keys: private and public (certificate).
3. Binary file signing (ELF):
   3.1 MD5 of the binary file is encrypted with the help of the private key;
   3.2 The encrypted MD5 is written to a new *.sig* section of the binary file;
   3.3 The certificate is saved to the ~/.ssh folder.

All this can be implemented with the help of the Linux utilities openssl, objcopy, and md5sum. Below you can find an example of a script sign_elf.sh that signs ELF binary files.

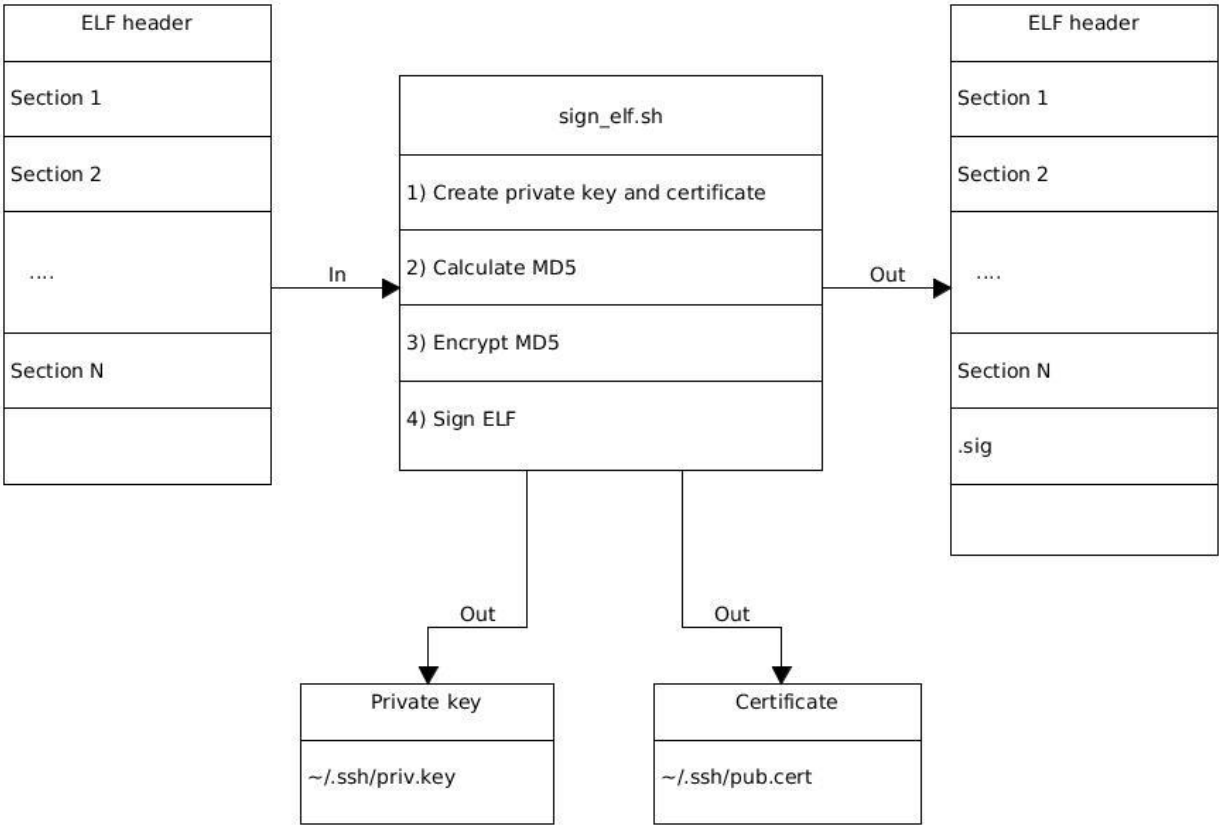

Figure 1. The process of signing ELF binary. (Source: Auriga)

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Let us explore the details of what this script does.

Line 19:

```
openssl req -nodes -x509 -sha256 -newkey rsa:4096 -keyout "$PRIVATE_KEY" -out
"$CERTIFICATE" -days 365 -subj "$SUBJECT"
```

req — certificate creation request

-nodes — create a private plaintext key

-x509 — output — self-signed certificate

-sha256 — encryption algorithm

-newkey rsa:4096 — create a new certificate and RSA private key, number of bits — 4096

-keyout $PRIVATE_KEY — the path to the file where the private key is written to

-out $CERTIFICATE — the path to the file where the certificate is written to

-days 365 — number of days for certificate acknowledgement

-subj $SUBJECT — new certificate subject (should have the format of /type0=value0/type1=value1/type2=…). In our case, this is /C=RU/ST=Nizhni Novgorod/L=Nizhniy Novgorod/O=Auriga/OU=DEV/CN=www.auriga.com, where

C — country

ST — state, region, province

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

L — location

O — organization

OU — organizational department/unit

CN — basic title/container name

The subject is described in detail in RFC-5280 (https://tools.ietf.org/html/rfc5280). After this command is run, a private key will be generated, ~/.ssh/priv.key and certificate ~/.ssh/pub.crt. The private key will be used to encrypt the data, and the certificate will be used for data decryption. Using one private key, it is possible to generate a number of unique certificates for decrypting data that was encrypted with this private key.

Line 21:

```
for ELF_BIN in $@; do
```

Start of loop for all binary files added to the sign_elf.sh script.

Line 33:

```
objcopy --remove-section=.sig "$ELF_BIN"
```

Remove the *.sig* section from our binary file. This needs to be conducted if the file was already signed with our script and we want to re-sign it.

Lines 36-39:

```
rm -f dummy.txt touch dummy.txt truncate --size=512 dummy.txt objcopy --add-
section .sig=dummy.txt --set-section-flags .sig=noload,readonly "$ELF_BIN"
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Create a 512-byte text file and add it to our binary file not loaded on the runtime *.sig* section only for reading, which contains data from the dummy.txt file.

Line 42:

```
md5sum "$ELF_BIN" | awk '{ print $1 }' > "$KEY_DIR/$ELF_BIN_MD5"
```

Calculate MD5 of the binary file (with *.sig* section) and write the result to a text file, binary_name.md5.

Line 45:

```
openssl dgst -sha256 -sign "$PRIVATE_KEY" -out "$KEY_DIR/$ELF_BIN_SIGNATURE"
"$KEY_DIR/$ELF_BIN_MD5"
```

This command encrypts the file with MD5 created by line 42 with a private key. Arguments:

dgst — this option indicates that we want to encrypt (sign) data;

-sha256 — encryption algorithm;

-sign $PRIVATE_KEY — encrypt the file with the help of private key $PRIVATE_KEY;

-out $KEY_DIR/$ELF_BIN_SIGNATURE — encrypted data is saved to file $KEY_DIR/$ELF_BIN_SIGNATURE;

$KEY_DIR/$ELF_BIN_MD5 — text file containing data to be encrypted.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Line 48:

```
openssl dgst -sha256 -verify <(openssl x509 -in "$CERTIFICATE" -pubkey -
noout) -signature "$KEY_DIR/$ELF_BIN_SIGNATURE" "$KEY_DIR/$ELF_BIN_MD5"
```

Signed file verification. It can be understood by reference to this line that for DS verification we need encrypted data, a certificate that will help us perform verification and data verification. That is, if

x — encrypted data,

y — certificate,

z — verification data,

then

f(x,y) = z

Line 52:

```
objcopy --update-section .sig="$KEY_DIR/$ELF_BIN_SIGNATURE" --set-section-
flags .sig=noload,readonly "$ELF_BIN" "$ELF_BIN"
```

Remove the old *.sig* section and add a new one to file $ELF_BIN (binary_name). As data for the new *.sig* section, data from the signed file $KEY_DIR/$ELF_BIN_SIGNATURE (~/.ssh/binary_name.sha256) is used.

Furthermore, let us review our DS verification method on runtime. We will use the libcrypto and libssl libraries for this and the following algorithm:

1. Retrieve the encrypted MD5 from the binary file;
2. Calculate the binary file MD5 with the *.sig* section filled out with nulls;

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

3. Verify that the encrypted MD5 equals the calculated MD5 with the help of a certificate.

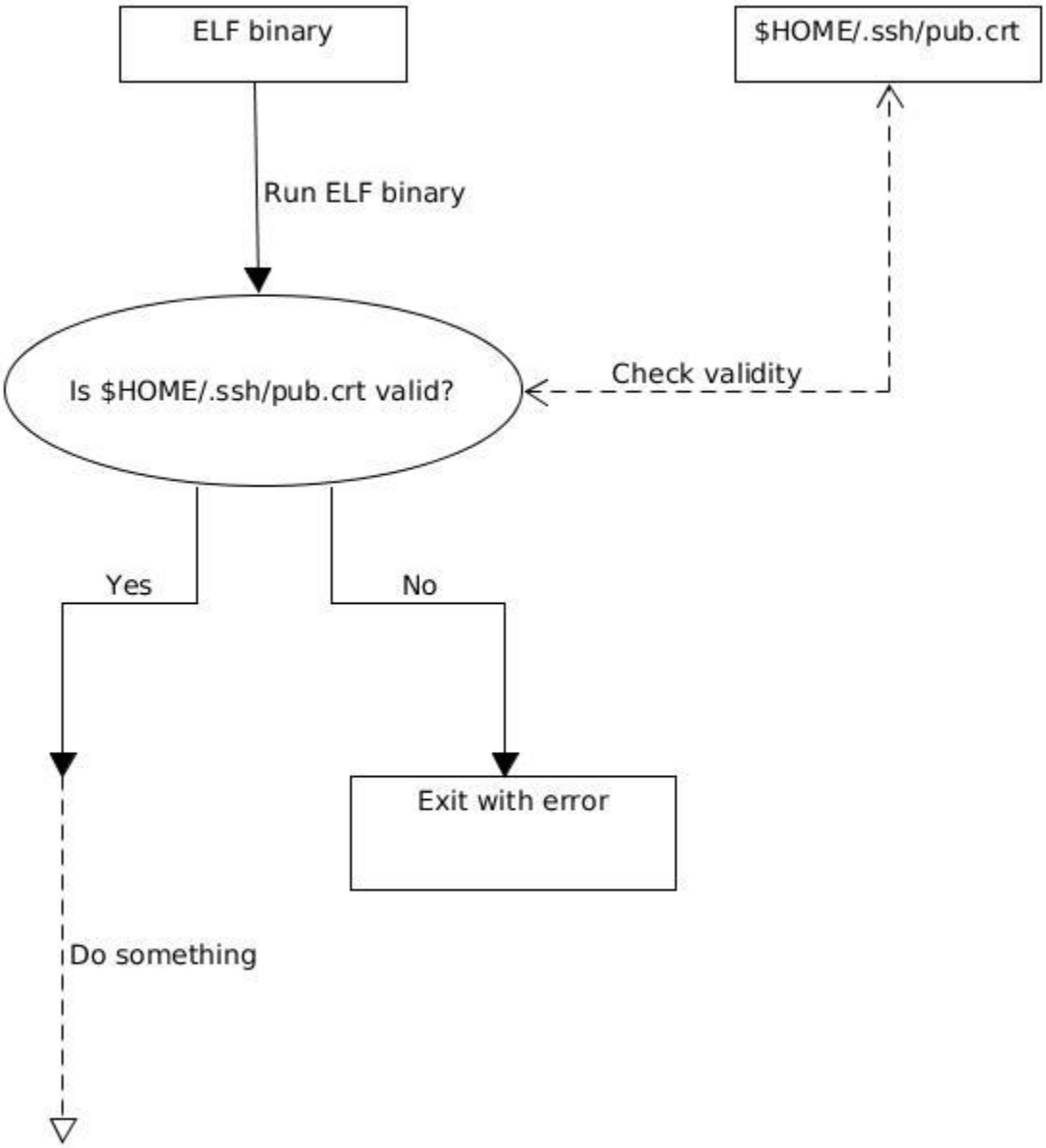

Figure 2. Check integrity. (Source: Auriga)

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

The main job will be performed by the check_integrity() function, which takes the path to the binary file as an argument and returns 0 if the signature verification is passed and (-1) otherwise. The function uses libcrypto library to verify DS. Let us examine the question of check_integrity() function operating in general:

Lines 236–243

```
if (getpwuid_r(geteuid(), &pws, buff, sizeof(buff), &result) != 0) {
        fprintf(stderr, "Failed to get home directory\n");
        goto cleanup;
}
```

Create a path to the certificate file. The current implementation expects that the file will be kept in the $HOME/.ssh ($HOME/.ssh/pub.crt) folder.

Line 246

```
OpenSSL_add_all_algorithms();
```

The OpenSSL_add_all_algorithms() function adds all algorithms to a special internal table that OpenSSL uses when calling different functions. Normally, this function is invoked at the beginning, and before exiting the application, EVP_cleanup() is invoked.

Line 249:

```
X509 *cert = X509_new();
```

Memory is allocated for the X509 structure, which is necessary for the X509 certificate introduction.

Line 256:

```
bio_cert = BIO_new_file(certificate, "rb");
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

The BIO object is created in association with the certificate file. The BIO object can be viewed as an analogue of the file stream returned by the fopen() function.

Line 263:

```
if (PEM_read_bio_X509(bio_cert, &cert, NULL, NULL) == NULL) {
```

Read from the certificate file into the X509 structure.

Line 269:

```
sigkey = X509_get_pubkey(cert);
```

Retrieve the public key from the certificate. The result is saved in the EVP_PKEY structure pointer.

Line 276:

```
mctx = EVP_MD_CTX_create();
```

Create the verification context (the result is moved to the EVP_MD_CTX structure). This structure will be used to verify our certificate.

Line 283:

```
if (!EVP_DigestVerifyInit(mctx, NULL, EVP_sha256(), NULL, sigkey)) {
```

Initialize the mctx verification context. The third parameter contains the encryption algorithm and the fifth — public key. The second and the fourth parameters are not needed for our verification — set NULL.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Line 288:

```
if (get_signature(binary, encrypted_md5)) {
```

Retrieve the encrypted MD5 from the binary file. The operation of this function will be described below in more detail.

Line 295:

```
siglen = EVP_PKEY_size(sigkey);
```

Calculate the maximum signature size in bytes.

Line 302:

```
if (calculate_md5(binary, md5)) {
```

Calculate MD5 of the binary file. The operation of this function will be described below in more detail.

Lines 308–315:

```
p = md5_string;
for (int i = 0; i < MD5_DIGEST_LENGTH; i++) {
     snprintf(p, MD5_DIGEST_LENGTH, "%02x", md5[i]);
     // one step is two symbols
     p = p + 2;
}
// Last symbol is new line
md5_string[MD5_STR_SIZE - 1] = '\n';
```

Convert MD5 returned by the calculate_md5() function into the line returned by the md5sum utility. We perform this conversion because the sign_elf.sh script encrypts MD5, which is output by the md5sum utility, and adds the encrypted MD5 into the *.sig* section.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Line 318:

```
EVP_DigestSignUpdate(mctx, md5_string, MD5_STR_SIZE);
```

Add the obtained MD5 as a line into the verification context.

Line 321:

```
ret = EVP_DigestVerifyFinal(mctx, encrypted_md5, (unsigned int)siglen);
```

This function performs the verification of the encrypted MD5 calculated on runtime MD5 (we remember that we have the public key and MD5 calculated on runtime in mctx). The EVP_DigestVerifyFinal() function returns a positive number if the verification is passed.

Now we move to the description of functions that work with the ELF header. The Get_signature() function parses the header of the 64-bit ELF file (the path to the file is set by the first parameter) and retrieves the encrypted MD5 from the *.sig* section. The encrypted MD5 pointer is saved in the second parameter. Let us provide more insight into this function.

Lines 148–153:

```
if (stat(fname, &st) != 0) {
    perror("stat");
    return 1;
}

size = st.st_size;
```

We obtain the file size in bytes.

Line 156:

```
fd = open(fname, O_RDONLY);
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Open the binary file for reading.

Line 163:

```
p = mmap(0, size, PROT_READ, MAP_PRIVATE, fd, 0);
```

Map the file to memory equal to the file size. After that, we can work with the file as with the memory.

After this, we will search for our *.sig* section.

Line 170:

```
ehdr = (Elf64_Ehdr *)p;
```

We obtain the ELF header pointer — the header is at the beginning of the file.

Line 173:

```
shdr = (Elf64_Shdr *)(p + ehdr->e_shoff);
```

We obtain a pointer to the sections table (our *.sig* section is somewhere among them).

Line 176:

```
shnum = ehdr->e_shnum;
```

Count the total sections number.

Line 179:

```
sh_strtab = &shdr[ehdr->e_shstrndx];
```

Obtain a pointer to the lines table section header. Here is the ehdr field e_shstrndx — this is the lines table index in the sections table.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Line 182:

```
sh_strtab_p = p + sh_strtab->sh_offset;
```

Obtain the lines table address from the beginning of the file. Here is the sh_offset field — the section shift from the beginning of the file.

Lines 185–189:

```
for (i = 0; i < shnum; ++i) {
    char *section_name = NULL;
    // Get section name
    section_name = (char *)(sh_strtab_p + shdr[i].sh_name);
```

Here the loop starts for all sections (shnum). Each section header has the sh_name field — this is line index in the lines table. To retrieve the section nam,e we need to address the offset symbols table sh_name.

Lines 192–201:

```
if (!strncmp(section_name, SIGNATURE_SECTION_NAME,
     strlen(SIGNATURE_SECTION_NAME))) {
   int section_size = 0;
   int section_offset = 0;

   // Get '.sig' section size
   section_size = shdr[i].sh_size;

   // Get '.sig' section offset from start of ELF binary file
   section_offset = shdr[i].sh_offset;
```

If the section name is *.sig*, then we obtain its size (the field sh_size of the section header) and offset from the beginning of the file (field sh_offset).

Line 204:

```
memcpy(encrypted_md5, (char *)(p + section_offset), section_size);
```

With the section beginning pointer and its size, copy the section content and encrypted_md5[] array. Now the array has the encrypted MD5. The Calculate_md5() function code is much like get_signature(). The Calculate_md5() function does the following:

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

1. Allocates memory equal to the signed binary file size;
2. Copies the mapped binary file to the allocated memory;
3. Searches for the .*sig* section;
4. Fills out the .*sig* section content with nulls;
5. Calculates MD5 for the retrieved memory content.

Section .*sig* is filled out with nulls to calculate MD5 for the binary file before it is signed. Here we need to be reminded of how we signed the binary file in the sign_elf.sh script:

```
touch dummy.txt truncate --size=512 dummy.txt objcopy --add-
section .sig=dummy.txt --set-section-flags .sig=noload,readonly "$ELF_BIN"
```

Here we created a 512-byte file and filled it out with nulls. After that, we the added .*sig* section into the binary file that contained the dummy.txt file data.

The DS implementation method described above can be improved: Fill out the .*sig* section not with nulls but with some known numbers, such as with the first 10 bytes from the .data section. To do this, you will need to slightly modify the calculate_md5() function.

The source implementation code is provided in **Annex 1**.

Conclusion

This article reviewed one of the variants for DS implementation in Linux. A strong hash function encrypted with a sender's private key lies at the root of the method we described. This implementation does not require writing a driver and is accomplished by standard Linux utilities. Another variant of DS support for Linux is DigSig (http://disec.sourceforge.net/), but, unfortunately, this project is no longer maintained. For more detailed information about DSs and asymmetric encryption, see the RFC pages: 5280 (https://tools.ietf.org/html/rfc5280), 7091 (https://tools.ietf.org/html/rfc7091), 3447 (https://tools.ietf.org/html/rfc3447), and 5959 (https://tools.ietf.org/html/rfc5959).

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Annex 1

main.c

```
#include <openssl/crypto.h>
#include <openssl/err.h>
#include <openssl/md5.h>
#include <openssl/pem.h>
#include <openssl/x509.h>
#include <stdio.h>
#include <unistd.h>

#include <elf.h>
#include <fcntl.h>
#include <pwd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>

#define MD5_STR_SIZE (MD5_DIGEST_LENGTH * 2 + 1)
#define SIGNATURE_SECTION_NAME ".sig"
#define FILE_PIECE (1024)
#define PWS_BUFFER (1024)
#define PATH_TO_CERTIFICATE "/.ssh/pub.crt"

void get_md5(char *m, size_t l, unsigned char *md5) {
        MD5_CTX mdContext = { 0 };
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
        MD5_Init(&mdContext);
        char *tmp = m;
        size_t bytes = l;

        // We cannot upload a big memory by one call of MD5_Update. Therefore, we
        // upload a whole file by pieces. The size of each piece is 1024 bytes,
        while (bytes > FILE_PIECE) {
                MD5_Update(&mdContext, tmp, FILE_PIECE);
                tmp = tmp + FILE_PIECE;
                bytes = bytes - FILE_PIECE;
        }

        // Upload last piece
        MD5_Update(&mdContext, tmp, bytes);

        // Calculate MD5
        MD5_Final(md5, &mdContext);
}

int calculate_md5(const char *const fname, unsigned char *md5) {
        struct stat st = {0};
        size_t size = 0;
        int fd = -1;
        int i = 0;
        int shnum = 0;
        const char *sh_strtab_p = NULL;
        char *p = NULL;
        char *m = NULL;
        Elf64_Ehdr *ehdr = NULL;
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
Elf64_Shdr *shdr = NULL;
Elf64_Shdr *sh_strtab = NULL;

// Get size of binary
if (stat(fname, &st) != 0) {
        perror("stat");
        return 1;
}

size = st.st_size;

// Open binary file for reading
fd = open(fname, O_RDONLY);
if (fd < 0) {
        perror("open");
        return 1;
}

// Map binary file
p = mmap(0, size, PROT_READ, MAP_PRIVATE, fd, 0);
if (p == MAP_FAILED) {
        perror("mmap");
        return 1;
}

// Allocate memory to store mapped file
m = (char *)calloc(size, sizeof(char));
if (m == NULL) {
        perror("calloc");
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
        munmap(p, size);
        return 1;
}

// Copy mapped binary file to allocated memory
memcpy(m, p, size);

// Unmap mapped file
munmap(p, size);

// Get pointer to ELF header
ehdr = (Elf64_Ehdr *)m;

// Get pointer to section header table
shdr = (Elf64_Shdr *)(m + ehdr->e_shoff);

// Get number of section header table items
shnum = ehdr->e_shnum;

// Get pointer to section header string table
sh_strtab = &shdr[ehdr->e_shstrndx];

// Get base address of section header string table
sh_strtab_p = m + sh_strtab->sh_offset;

// For each section
for (i = 0; i < shnum; ++i) {
        char *section_name = NULL;
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
            // Get section name
            section_name = (char *)(sh_strtab_p + shdr[i].sh_name);

            // If it is '.sig' section
            if (!strncmp(section_name, SIGNATURE_SECTION_NAME,
                                    strlen(SIGNATURE_SECTION_NAME))) {
                // Fill section content with zeros
                memset(m + shdr[i].sh_offset, 0, shdr[i].sh_size);
            }
        }

        // Calculate MD5 of memory
        get_md5(m, size, md5);

        // Free memory
        free(m);

        munmap(p, size);

        return 0;
}

int get_signature(const char *const fname, unsigned char *encrypted_md5) {
        Elf64_Ehdr *ehdr = NULL;
        Elf64_Shdr *shdr = NULL;
        Elf64_Shdr *sh_strtab = NULL;
        struct stat st = {0};
        size_t size = 0;
        int fd = -1;
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
int i = 0;
int shnum = 0;
const char *sh_strtab_p = NULL;
char *p = NULL;

// Get size of binary
if (stat(fname, &st) != 0) {
        perror("stat");
        return 1;
}

size = st.st_size;

// Open binary file for reading
fd = open(fname, O_RDONLY);
if (fd < 0) {
        perror("open");
        return 1;
}

// Map binary file
p = mmap(0, size, PROT_READ, MAP_PRIVATE, fd, 0);
if (p == MAP_FAILED) {
        perror("mmap");
        return 1;
}

// Get pointer to ELF header
ehdr = (Elf64_Ehdr *)p;
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
// Get pointer to section header table
shdr = (Elf64_Shdr *)(p + ehdr->e_shoff);

// Get number of section header table items
shnum = ehdr->e_shnum;

// Get pointer to section header string table
sh_strtab = &shdr[ehdr->e_shstrndx];

// Get base address of section header string table
sh_strtab_p = p + sh_strtab->sh_offset;

// For each section
for (i = 0; i < shnum; ++i) {
        char *section_name = NULL;

        // Get section name
        section_name = (char *)(sh_strtab_p + shdr[i].sh_name);

        // If it is '.sig' section
        if (!strncmp(section_name, SIGNATURE_SECTION_NAME,
                                strlen(SIGNATURE_SECTION_NAME))) {
                int section_size = 0;
                int section_offset = 0;

                // Get '.sig' section size
                section_size = shdr[i].sh_size;
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
                        // Get '.sig' section offset from start of ELF binary file
                        section_offset = shdr[i].sh_offset;

                        // Copy content of '.sig' section to array
                        memcpy(encrypted_md5, (char *)(p + section_offset), section_size);
                }
        }

        munmap(p, size);
        return 0;
}

int check_integrity(const char * const binary) {
        EVP_MD_CTX *mctx = NULL;
        EVP_PKEY *sigkey = NULL;
        BIO *bio_cert = NULL;
        struct passwd *result = NULL;
        struct passwd pws = {0};
        char md5_string[MD5_STR_SIZE] = {0};
        char certificate[PATH_MAX] = {'\0'};
        unsigned char encrypted_md5[512] = {0};
        unsigned char md5[MD5_DIGEST_LENGTH] = {0};
        char buff[PWS_BUFFER] = {0};
        char *p = NULL;
        int siglen = 0;
        int ret = -1;

        if ((binary == NULL) || (binary[0] == '\0')) {
                fprintf(stderr, "Binary file is not valid\n");
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
        goto cleanup;
}

// Get home directory. We call getpwuid_r() instead of getpwuid() because
// getpwuid_r() is thread-safe. Also we call geteuid() instead of getuid()
// to get effective user ID: only user who is owner of executable file
// has to be able to run this file
if (getpwuid_r(geteuid(), &pws, buff, sizeof(buff), &result) != 0) {
        fprintf(stderr, "Failed to get home directory\n");
        goto cleanup;
}

// Create absolute path for certificate
strncpy(certificate, pws.pw_dir, strlen(pws.pw_dir));
strncat(certificate, PATH_TO_CERTIFICATE, strlen(PATH_TO_CERTIFICATE));

// Add all digest algorithms to the table
OpenSSL_add_all_algorithms();

// allocates and initializes a X509 object
X509 *cert = X509_new();
if (cert == NULL) {
        fprintf(stderr, "X509_new() failed\n");
        goto cleanup;
}

// Create BIO object associated with certificate
bio_cert = BIO_new_file(certificate, "rb");
if (bio_cert == NULL) {
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
        fprintf(stderr, "BIO_new_file() failed\n");
        goto cleanup;
}

// Read certificate in PEM format from BIO
if (PEM_read_bio_X509(bio_cert, &cert, NULL, NULL) == NULL) {
        fprintf(stderr, "PEM_read_bio_X509 failed\n");
        goto cleanup;
}

// Get public key from certificate
sigkey = X509_get_pubkey(cert);
if (bio_cert == NULL) {
        fprintf(stderr, "X509_get_pubkey() failed\n");
        goto cleanup;
}

// Create message digest context
mctx = EVP_MD_CTX_create();
if (mctx == NULL) {
        fprintf(stderr, "EVP_MD_CTX_create() failed\n");
        goto cleanup;
}

// Set up verification context mctx using public key
if (!EVP_DigestVerifyInit(mctx, NULL, EVP_sha256(), NULL, sigkey)) {
        fprintf(stderr, "EVP_DigestVerifyInit() failed\n");
        goto cleanup;
}
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
// Get encrypted signature from ELF binary
if (get_signature(binary, encrypted_md5)) {
        fprintf(stderr, "get_signature() failed\n");
        goto cleanup;
}

// Get sigkey size
siglen = EVP_PKEY_size(sigkey);
if (siglen <= 0) {
        fprintf(stderr, "Error reading signature file\n");
        goto cleanup;
}

// Get original MD5 from ELF
if (calculate_md5(binary, md5)) {
        fprintf(stderr, "get_signature() failed\n");
        goto cleanup;
}

// Convert MD5 digital to human readable string
p = md5_string;
for (int i = 0; i < MD5_DIGEST_LENGTH; i++) {
        snprintf(p, MD5_DIGEST_LENGTH, "%02x", md5[i]);
        // one step is two symbols
        p = p + 2;
}
// Last symbol is new line
md5_string[MD5_STR_SIZE - 1] = '\n';
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
        // Add buffer (original MD5) to be compared to context
        EVP_DigestSignUpdate(mctx, md5_string, MD5_STR_SIZE);

        // Add encrypted buffer to context and perform verification
        ret = EVP_DigestVerifyFinal(mctx, encrypted_md5, (unsigned int)siglen);
        if (ret > 0) {
                fprintf(stderr, "Verified OK\n");
                ret = 0;
        } else if (ret == 0) {
                fprintf(stderr, "Verification Failure\n");
                ret = -1;
        } else {
                fprintf(stderr, "Error Verifying Data\n");
                ret = -1;
        }

cleanup:
        // Release objects
        EVP_MD_CTX_destroy(mctx);
        X509_free(cert);
        EVP_PKEY_free(sigkey);
        BIO_free(bio_cert);
        EVP_cleanup();

        return ret;
}

int main(int argc, char **argv) {
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
        if (check_integrity(argv[0]) < 0) {
                fprintf(stderr, "Signature check failed\n");
                return 1;
        }

        /* Do something here */

        return 0;
}
```

Makefile

```
CC = gcc
RM = rm -f

TARGET = main
OBJS = main.o
LIBS= -lcrypto -lssl
FLAGS = -Wall -pedantic -fsanitize=address -lasan

%.o: %.c
        $(CC) $(FLAGS) -c $^ -o $@

all: $(TARGET)

$(TARGET): $(OBJS)
        $(CC) $(FLAGS) -o $@ $^ $(LIBS)
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
clean:
	$(RM) $(OBJS) $(TARGET)
```

README

```
------------------------
1) How to build
$ make clean
$ make all
------------------------
2) How to sign:
./sign_elf.sh main
------------------------
3) How to run:
$ ./main
```

sign_elf.sh

```
#!/bin/bash

KEY_DIR="$HOME/.ssh"
PRIVATE_KEY="$KEY_DIR/priv.key"
CERTIFICATE="$KEY_DIR/pub.crt"
SUBJECT="/C=RU/ST=Nizhni Novgorod/L=Nizhniy
Novgorod/O=Auriga/OU=DEV/CN=www.auriga.com"

if [ "$#" = "0" ]; then
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
        echo "Usage: sign_elfs.sh <elf file1> <elf file2> ... <elf fileN>"
        exit 1;
fi

if [ ! -d "$KEY_DIR" ]; then
        # Control will enter here if $DIRECTORY does not exist.
        mkdir "$KEY_DIR"
fi

# Create private key and certificate
openssl req -nodes -x509 -sha256 -newkey rsa:4096 -keyout "$PRIVATE_KEY" -out
"$CERTIFICATE" -days 365 -subj "$SUBJECT"

for ELF_BIN in $@; do
        ELF_BASE_NAME="${ELF_BIN##*/}"
        #       ELF_BIN_OLD="$ELF_BIN.old"
        ELF_BIN_SIGNATURE="$ELF_BASE_NAME.sha256"
        ELF_BIN_MD5="$ELF_BASE_NAME.md5"

        if [ ! -f "$ELF_BIN" ] || [ "x$ELF_BIN" = "x" ];then
                echo "Error: no such file $ELF_BIN"
        exit 1
        fi

        # Remove .sig section
        objcopy --remove-section=.sig "$ELF_BIN"
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
# Add 512-byte section filled with zeros
rm -f dummy.txt
touch dummy.txt
truncate --size=512 dummy.txt
objcopy --add-section .sig=dummy.txt --set-section-flags .sig=noload,readonly
"$ELF_BIN"

# Create MD5 hash
md5sum "$ELF_BIN" | awk '{ print $1 }' > "$KEY_DIR/$ELF_BIN_MD5"

# Encrypt MD5 hash using private key
openssl dgst -sha256 -sign "$PRIVATE_KEY" -out
"$KEY_DIR/$ELF_BIN_SIGNATURE" "$KEY_DIR/$ELF_BIN_MD5"

# Validate encrypted MD5 hash using certificate
openssl dgst -sha256 -verify <(openssl x509 -in "$CERTIFICATE" -pubkey -noout) -
signature "$KEY_DIR/$ELF_BIN_SIGNATURE" "$KEY_DIR/$ELF_BIN_MD5"

# Add encrypted MD5 hash into ELF binary into .sig section
echo "Add .sig section"
objcopy --update-section .sig="$KEY_DIR/$ELF_BIN_SIGNATURE" --set-section-
flags .sig=noload,readonly "$ELF_BIN" "$ELF_BIN"

# Print .sig section
echo "Check .sig section"
objdump -sj .sig "$ELF_BIN"
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
done

rm -f dummy.txt

ls -ls ~/.ssh
```