Flashback: Licensing Malware To Hinder Analysis and Functionality

Author: Daryl Ashley, ashley@infosec.utexas.edu
Senior Network Security Analyst
April 24, 2012

Abstract

License keys have been used for a number of years to prevent the unauthorized use of a number of software packages. The authors of the Flashback Trojan appear to have incorporated a licensing technique during installation of the trojan that ensures that the binary installed on an infected Mac OS X computer cannot be run on other computers, including in a sandbox environment. This paper describes the licensing technique and some other obfuscation techniques used by the trojan.
Table of Contents

INTRODUCTION ................................................................................................................................. 4
OVERVIEW OF FUNCTIONALITY ......................................................................................................... 4
BLOB RECORD STRUCTURE AND DATA DECRYPTION ................................................................. 6
DYNAMIC RESOLUTION OF SYSTEM CALL ADDRESSES ............................................................... 9
DEOBFUSCATION OF HTTP COMMUNICATION SUBROUTINE ..................................................... 11
INSTALLATION OF LICENSED VERSION ......................................................................................... 12
ENCRYPTION OF THE BLOB FIELD ............................................................................................... 16
CONCLUSION .................................................................................................................................. 17
BIBLIOGRAPHY ................................................................................................................................. 18
APPENDIX 1: BLOB EXTRACTION CODE .......................................................................................... 19
APPENDIX 2: BLOB RECORD EXTRACTION CODE ......................................................................... 20
APPENDIX 3: SYSTEM CALL TO VARIABLE MAPPINGS .................................................................. 23
Introduction

Dr. Web, a Russian anti-virus vendor, recently reported that 550,000 Mac OS X computers have been infected with a trojan known as Flashback (Cheng 2012). Shortly after Dr. Web released the information, several other security companies confirmed the extent of the infection (Albanesius 2012, Brod 2012) and provided analysis of the malware (Gostev 2012). The trojan is of great interest because it calls into question the security of Mac OS X computers (Kerner 2012).

The malware is also interesting because it uses several techniques to hinder analysis of itself. Among those techniques are encryption and obfuscation of system calls. Obfuscation of system calls using the dlopen and dlsym functions, as well as the encryption of data within binaries, have been documented in other papers (Zhao et al.). However, the Flashback Trojan may be unique because it appears to install a “licensed” version of the original malware in addition to using standard obfuscation techniques. During the installation of the licensed version, the infected host’s UUID is used to encrypt some data within the original binary that is needed for the malware to function properly. When the licensed version of the malware is executed, the infected host’s UUID is used to decrypt the information. Without the host’s UUID, the licensed version will not be able to decrypt the data and the malware will not function properly. This prevents the malware from being analyzed on a computer that does not have the correct UUID, so analysis via a sandbox environment is prevented. Static analysis of the licensed version without the infected host’s UUID is also difficult because the encrypted data contains the names of the libraries and system calls that are resolved using dlopen and dlsym.

This paper describes how the Flashback Trojan installs a licensed version of itself on the infected host, using the infected host’s UUID as a seed in the licensing process. This paper also shows how obfuscated system calls can be resolved once the data is decrypted. The md5 checksum of the unlicensed binary is 5ee8b7333f1dee03f1c5f63b3f596e24 and the md5 checksum of the licensed binary is fae406de8d8516744efc4fe6eb37ca8. The binary that is analyzed was identified as a “Mach-O universal binary with 2 architectures”, by the “file” command. The binary contains both a 32-bit and 64-bit version of the malware. The original and licensed specimens are available upon request from the author.

Overview of Functionality

The malware specimen analyzed in this paper was included in a jar file that attempts to exploit a host that is vulnerable to a Java exploit (Brod 2012). Analysts have shown that the malware will not run if certain applications, such as Little Snitch or ClamXav are present on the host (Gostev 2012). If these applications are not present, the malware will continue to execute.

The malware specimen contains a chunk of binary data that is 0x1051 bytes in length. This binary data is located at offset 0x510C when the malware is analyzed using IDA Pro. This paper will refer to this binary chunk of data as the “blob” field. The blob contains a series of records. A number of the records
contain information needed by the malware to function properly. The data in a blob record must undergo two separate decryption steps before the data can be used by the malware. The blob records store information such as the names of libraries that are loaded dynamically via calls to `dlopen` and names of functions whose addresses must be resolved via calls to `dlsym`. The blob record format will be described later.

If the malware executes properly, the following occurs after decryption of the blob data:

1. The host sends an HTTP GET request to a URL that is hard-coded in the executable
2. The host generates a cipher key using the UUID as a seed for the key generation algorithm
3. The blob field is encrypted with the key generated in step 2.
4. The blob field in the original malware binary is overwitten with the encrypted blob
5. The hard-coded URL in the malware binary from step 1 is zeroed out.

At this point, the original malware binary has been replaced with a blob field that has been encrypted using a key generated by an algorithm using the host's UUID as a seed for the algorithm. The algorithm used is symmetric, so the same algorithm is used to decrypt the blob when the new executable is run. In this way, the new executable has been “licensed”. If the executable is run on a different host, the UUID will generate a different key, and the blob field will not be decrypted properly.

When the new binary is executed, the following occurs:

1. The blob field is decrypted using the key generated by the host UUID
2. The record data is decrypted so that the malware has all information needed for its functionality
3. The host enters into a Command and Control (C&C) loop

Note that the URL that was hard-coded in the original binary is no longer present, so the URL is not contacted again. An overview of the malware's functionality is provided below in the following C style snippet:
#include <...>

char url[] = "http://phonehomesite.com/stat_svc/";
char blob[] = "Encrypted stuff...";
uint32_t blob_length = 0x1051;

int main ()
{
    determine_machine_uid();

    if (url) {
        decrypt_blob_record_with_pipe_delimited_libraries();
        decrypt_blob_record_with_pipe_delimited_system_call_names();
        use_dlopen_and_dlsym_to_obtain_system_call_addresses();

        send_http_get_request_to_url();
        copy_exe_to_memory_buffer();
        zero_out_url_in_memory_buffer();
        generate_cipher_key_using_uuid_as_seed();
        encrypt_blob_using_cipher_key();
        replace_blob_in_memory_with_encrypted_blob();
        overwrite_exe_with_contents_of_memory();
    }
    else {
        generate_cipher_key_using_uuid_as_seed();
        decrypt_blob_using_cipher_key_above();

        decrypt_blob_record_with_pipe_delimited_libraries();
        decrypt_blob_record_with_pipe_delimited_system_call_names();
        use_dlopen_and_dlsym_to_obtain_system_call_addresses();

        do_C&C_stuff()
    }
    return 0;
}

Blob Record Structure and Data Decryption

Figure 1 shows a portion of the blob field embedded in the malware specimen. A C program that can be used to extract the contents of the blob field from the malware binary is included in Appendix 1. Each record in the blob field has the following layout:

<table>
<thead>
<tr>
<th>Position</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Signature</td>
<td>always 0xFD</td>
</tr>
<tr>
<td>0x1-0x2</td>
<td>Key</td>
<td>a unique key value to identify the record</td>
</tr>
<tr>
<td>0x3-0x6</td>
<td>Data length</td>
<td>length of the data</td>
</tr>
<tr>
<td>0x7-...</td>
<td>Data</td>
<td>encrypted data</td>
</tr>
</tbody>
</table>

The value at offset 0x5120 in the figure is 0xFD. This is the signature portion of the record. The next two bytes are 0x6192 in little endian format. These bytes are the record key. The next four bytes are 0x0000000C in little endian format. This is the length of the data stored in the record. Finally, the 0xC bytes starting at offset 0x5127 is the record data. To determine the location of the next record, add the record length to the offset of the data (0x5127 + 0xC = 0x5133). Note that the value located at 0x5133 in
the figure is 0xFD. The blob can be searched for a record with a specific key value by iterating in this way through the blob.

| 00005120 | FD | 92, 61, 0C, 60, 00, 00, 00, F0 | AA, 40, 53, 99, CC, AC, 8A, 20 |
| 00005130 | C7, 2B, 60, FD, 8F, D1 | 00, 00, 8B, A2, AD, 02, 04 |
| 00005140 | A4, D4, 29, BC, 94, BD, EB, CB | 1C, FE, 13, 7D, BA, F8, 2A, C5 |
| 00005150 | 73, 79, DF, 7C, 6B, 3A, 16, AB | 2C, 96, 5B, CC, 40, 71, E5 |
| 00005160 | 02, A2, E8, F0, CB | 48, D3, BF, 73, 95, 8B, AC, 57, 91, 8E, 50 |
| 00005170 | 4C, A4, 04, D1, 8E, EC | 42, 18, 9C, 43, FD, B7, A1, AA, F2, D7 |
| 00005180 | 2C, AE, D1, ED | 3E, FB, 76, 7D, 81, 26, F4, 6A, B7, 6C, C8, 7C |

Figure 1: Blob embedded in Flashback Trojan

The malware extracts record data from the blob field by making calls to subroutine sub_4A11. Figure 2 shows the subroutine being called to retrieve the record with key 0x6192, the key of the first record shown in Figure 1. The subroutine is also passed two 32 bit values, 0x27354581 and 0xA2937647. These two values are used to decrypt the record data. The decrypted data is written to a memory buffer that can be accessed by var_1C. The length of the memory buffer is stored in var_20. Similar calls are made at a number of locations in the subroutine sub_23EF. The blob record with the key value 0xF12E contains the library names needed by the malware. The data for this record is stored in a memory buffer that can be accessed via var_28 after sub_4A11 is called at offset 0x2824. The blob record with the key value of 0xE002 contains the function names needed by the malware. The data for this record is stored in a memory buffer that can be accessed via var_30.

| 0000097C | lea | edx, [ebp+var_1C] |
| 0000097F | lea | eax, [ebp+var_20] |
| 00000982 | mov | [esp+10h], edx |
| 00000986 | mov | [esp+14h], eax |
| 0000098A | mov | dword ptr [esp+8], 27354581h |
| 00000992 | mov | dword ptr [esp+8Ch], 0A2937647h |
| 00000994 | mov | dword ptr [esp+4], 6192h |
| 0000099A | mov | [esp], edx |
| 0000099E | call | sub_4A11 |

Figure 2: Calling sub_4A11 to extract record data from the blob field

The first record in the blob is special because it contains three 32-bit values that are used as keys in a second decryption step for the remaining records in the blob. Figure 3 shows the three 32-bit values being stored in the variables var_1FBC, var_1FC0, and var_1FC4. Figure 4 shows the three variables and var_28 being passed as parameters to subroutine sub_2296. This subroutine uses the three variables to perform the second decryption step on the memory buffer that is accessed via var_28. The second decryption step produces the following pipe delimited list of library names:

```
/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit/System/Library/Frameworks/CoreServices.framework/Versions/A/CoreServices/usr/lib/libgcc_s.1.dylib/usr/lib/libbz.dylib/usr/lib/libssl.dylib/usr/lib/libcrypto.dylib
```

/Securus // Vigilare // Insanus
Figure 3: Data from record with key 0x6192 is stored in three variables.

Figure 4: Call to second decryption routine, sub.2296.
Dynamic Resolution of System Call Addresses

Two decrypted blob records are used to resolve the addresses of system calls. The first is a pipe-delimited list of library names. The second is a pipe-delimited list of system call names. The list of library names was shown in the previous section. The list of system call names is shown below.

\_NSGetExecutablePath|CFStringCreateWithCString|CFStringGetCString|CFRelease|CFURLCreateWithString|CFHTTPMessageCreateRequest|CFHTTPMessageSetHeaderFieldValue|CFReadStreamCreateForHTTPRequest|CFReadStreamOpen|CFReadStreamRead|CFReadStreamClose|IORegistryEntryFromPath|IORegistryEntryCreateCFProperty|IOObjectRelease|uncompress|compressBound|compress2|__CFStringMakeConstantString|BIO_new|BIO_crl|BIO_write|BIO_free_all|BIO_push|BIO_new_mem_buf|BIO_f_base64|BIO_s_mem|BIO_read|RSA_verify|SHA1|gethostbyname|BN_bin2bn|RSA_{new

The library names are parsed using the `strtok_r` function. The `strdup` function is used to create a duplicate buffer containing each library name. Each duplicated buffer is added to a linked list by calling subroutine sub_46DD. A head node pointing to the first entry of the linked list can be accessed via the variable `off_617C`. Figure 5 shows the portion of the malware that loops through the pipe-delimited list of library names.

The function names are also parsed using the `strtok_r` function. However, after duplicating each function name using the `strdup` function, the address of each duplicated function name is stored in an array. The base of the array is accessed using the variable named "symbol" by IDA Pro. Figure 6 shows the portion of the malware that creates the entries in the symbol array.

The subroutine named sub_1F54 is used to dynamically load any libraries needed in the linked list using the `dlopen` system call. The subroutine also uses the `dlsym` system call to resolve each system call name to the address that the system call is loaded in memory. The address is stored in a global variable that is used to access the system call in other portions of the malware. Figure 7 shows the calls made to subroutine sub_1F54 and Figure 8 shows the calls to `dlopen` and `dlsym` made in sub_1F54. A table showing the system call names and the variables that store the addresses is shown in Appendix 3.
information security office | the university of texas at austin

Figure 5: Using strtok_r to parse the pipe delimited list of library names

Figure 6: Using strtok_r and strdup to parse the pipe delimited list of function names

Figure 7: Resolving system call addresses
Figure 8: Calls to dlopen and dlsym

Deobfuscation of HTTP Communication Subroutine

Figure 9 shows a portion of subroutine sub_1FB3. The subroutine makes calls to function addresses that are stored in several global variables. The obfuscation of the system calls makes it difficult to determine the purpose of the subroutine. However, once the global variables are cross-referenced with the system call names, it is easier to see that this subroutine is used to send HTTP GET requests to a remote site.

Figure 9: Obfuscated subroutine
Installation of Licensed Version

Figure 10 shows the variable used by the malware to access the URL that is contacted during the initial execution of the malware. The figure also shows the variables used to determine the length and location of the blob field. These variables are named off_5108 (length of blob field), off_5104 (URL), and unk_5120 (blob field). The figure also shows the variable that stores the host's UUID (off_6184). Figure 11 shows the URL that is accessed using the variable off_5104.

Figure 10: Some important variables

When the malware is executed, subroutine sub_1FB3 is used to send an HTTP GET request to the URL that is accessed through off_5104. Figure 12 shows the call to sub_1FB3 at program offset 0x3264. Note that at program offset 0x323A, the HTTP GET request will not be sent if no URL is present in the malware binary. This is the portion of the malware in which branching to the installation of a licensed version or execution of the C&C loop occurs. Also note that the malware determines its fully qualified path and name using the NSGetExecutablePath system call at offset 0x322C.
After the HTTP GET request is sent to the C&C site, the malware starts to overwrite the original malware binary with a licensed version of itself. Figure 13 shows calls to `fopen`, `fseek`, and `ftell`. These system calls are used to open the malware binary in read only mode, set the file pointer to the end of the file, and determine the position of the file pointer. In this way, the malware is able to determine the size of the binary. Once the size of the binary is determined, the `malloc` system call is used to allocate a memory buffer that can be used to store a copy of the binary. Then a call is made to `fread` to copy the binary into the memory buffer.

Figure 14 shows the portion of the malware that is used to zero out the URL that is accessed via `off_5104`. The memory buffer containing the copy of the binary is examined and zeroed out using the `memcmp` and `memset` system calls. Recall that this binary contains both a 32-bit and 64-bit version of the malware. So, there should be two copies of the URL within the binary. The memory buffer is examined one byte at a time until the entire memory buffer has been examined, so both URLs will be zeroed out.

The instructions at offsets 0x3397 - 0x34CD are used to create an encrypted copy of the blob field. The algorithm used is examined in more detail in the next section. Once the blob field has been encrypted, the `memcmp` and `memset` system calls are used to replace the original blob fields in the memory buffer. The logic is similar to that used to zero out the URLs in Figure 14. After the blob fields in the memory buffer have been replaced with encrypted versions, the original binary is overwritten with the contents of the memory buffer. Figure 15 shows the overwritten of the original blob field in the memory buffer and Figure 16 shows the original binary being overwritten.
Figure 13: Copying malware binary to allocated memory buffer

```
xtext:0000350 loc_3350:
 xtext:0000355 loc_3355:
 xtext:0000350 loc_3350:
```

Figure 14: Searching for and zeroing out URL string

```
xtext:0000356 loc_3356:
```

Information Security Office | The University of Texas at Austin

SECURUS // VIGILARE // INSANUS
Figure 15: Searching for and replacing the original blob field

Figure 16: Overwriting the original binary with the watermarked version
Encryption of the Blob Field

The encryption of the blob field can be broken up into two steps. The first step is the generation of an encryption key and the second step is the actual encryption of the blob field. A C code snippet that shows the generation of the encryption key is shown in Figure 17. The infected host’s UUID string is used as a seed for an algorithm that uses a series of transpositions to produce a 256-byte key.

A C code snippet that shows the encryption process is shown in Figure 18. The algorithm is used to generate a stream of bytes that are XOR’d with the blob field. This algorithm is similar to the one-time tape described in (Schneier 2006), with the exception that the stream of bytes is not completely random. One important characteristic of this algorithm is that it is symmetric. The algorithm used for encryption is also used to decrypt the blob record.

```c
unsigned char alphabet[256];
unsigned char uuid[] = "00000000-0000-1000-8000-000C29074429";
CreateKey(alphabet, uuid, strlen(uuid));

void CreateKey (unsigned char *alphabet, unsigned char *seed, uint32_t length)
{
    uint32_t i, j, pos;
    uint64_t val;
    unsigned char val1, val2;
    val1 = 0;
    val2 = 0;
    for (i = 0; i < 256; i++) {
        alphabet[i] = i;
    }
    for (i = 0; i < 256; i++) {
        val = i;
        val >>= 0x1F;
        val <<= 32;
        val += i;
        pos = val % length;
        val1 = alphabet[i];
        val2 = (val1 + val2 + seed[pos]) & 0xFF;
        // swap values
        alphabet[i] = alphabet[val2];
        alphabet[val2] = val1;
    }
    return;
}

uint32_t idx1 = 0;
uint32_t idx2 = 0;
for (i = 0; i < BLOB_SIZE; i++) {
    idx1++;
    idx1 &= 0xFF;
    idx2 += alphabet[idx1];
    idx2 &= 0xFF;
    unsigned char val = alphabet[idx1];
    alphabet[idx1] = alphabet[idx2];
    alphabet[idx2] = val;
    uint32_t pos = alphabet[idx1] + alphabet[idx2];
    pos &= 0xFF;
    bin[i] ^= alphabet[pos];
}
```

Figure 17: Key Generation Algorithm

Figure 18: One-time tape
The instructions at offsets 0x2541 – 0x2659 are identical in functionality to the instructions at offsets 0x3397 – 0x34CD. Figure 10 shows that a jump is made to offset 0x2541 if no URL is hard-coded in the malware binary. This portion of the code is used to decrypt the blob field in the licensed version of the malware.

Conclusion

A number of automated sandbox environments have been developed to help safely analyze malware specimens. The sandbox environments run the malware in a controlled environment and observe the behavior of the malware. These tools provide a quick and safe method for analyzing and reporting about the behavior of a specific malware specimen. However, the authors of the Flashback Trojan may have taken a step to combat the use of sandbox environments for analysis. In doing so, they have reduced the tool set available to security analysts that are interested in studying their behavior.
Bibliography

Albanesius, Chloe (2012, April 6). Kaspersky Confirms Widespread Mac Infections Via Flashback Trojan. Retrieved April 24, 2012 from PCMag Website:
http://www.pcmag.com/article2/0,2817,2402715,00.asp


Cheng, Jacqui (2012, April 4). Flashback Trojan reportedly controls half a million Macs and counting. Retrieved April 24, 2012 from ars technica Website:


Appendix 1: Blob Extraction Code

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#define FILE_SIZE 59844
#define CIPHER_SIZE 0x1051

int patternMatch (uint8_t *buf, uint8_t *pattern, int length);

int main ()
{
    FILE *p, *in, *out;
    uint8_t cipher[CIPHER_SIZE];
    uint8_t exe[FILE_SIZE];
    unsigned int tmp;
    int patternLength;
    int i, result;
    uint8_t pattern[] = {
        0xFD, 0x92, 0x61, 0x0C, 0x00, 0x00, 0x00, 0xF0, 0xAA, 0x40, 0x53,
        0x99, 0xCC, 0xAC,
        0x8A, 0x20
    };
    patternLength = 16;

    in = fopen("sbm", "r");
    if (!in) {
        puts("Couldn't open sbm file");
        return 1;
    }

    result = fread (exe, 1, FILE_SIZE, in);
    fclose(in);
    int matchSpot = 0;
    for (i = 0; i < (FILE_SIZE - patternLength - 100); i++) {
        if (patternMatch(&exe[i], pattern, patternLength)) {
            printf("Pattern match found at: %i\n", i);
            matchSpot = i;
            break;
        }
    }

    if (matchSpot) {
        for (i = 0; i < CIPHER_SIZE; i++) {
            cipher[i] = exe[matchSpot + i];
        }
    }

    out = fopen("out.bin", "w");
    if (!out) {
        puts("Couldn't open output file");
        return 1;
    }

    result = fwrite(cipher, 1, CIPHER_SIZE, out);
    fclose(out);

    return 0;
}

int patternMatch (uint8_t *buf, uint8_t *pattern, int length)
{
    int i;
    for (i = 0; i < length; i++) {
        if (buf[i] != pattern[i]) {
            return 0;
        }
    }

    return 1;
}
```

Appendix 2: Blob Record Extraction Code

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <inttypes.h>

#define FILE_SIZE 0x1051

struct cipher_key {
    uint32_t key1;
    uint32_t key2;
};

struct cipher_key2 {
    uint32_t key1;
    uint32_t key2;
    uint32_t key3;
};

unsigned char *getBlock (unsigned char *buf, int *size, uint16_t match, int length, struct cipher_key *key);
unsigned char decryipt_byte (uint32_t key1, uint32_t key2, uint32_t key3, uint16_t in);
void decrypt_block(struct cipher_key *key, struct cipher_key2 *key2, uint16_t sig);

unsigned char bin[FILE_SIZE];

int main (int argc, char **argv) {
    FILE *in;
    unsigned char i;
    unsigned char *record;
    int size, result;
    struct cipher_key cipher_key;
    struct cipher_key2 cipher_key2;
    uint32_t record_key;

    printf("Number of arguments: %i
", argc);
    if (argc != 4) {
        puts("Usage: decrypt_record [record key] [key 1] [key 2]");
        return 1;
    }

    in = fopen("cipher.bin", "r");
    if (!in) {
        puts("Couldn't open cipher file");
        return 1;
    }

    result = fread (bin, 1, FILE_SIZE, in);
    fclose(in);

    cipher_key.key1 = 0x27354581;
    cipher_key.key2 = 0xA2937647;
    record = getBlock(bin, &size, 0x6192, FILE_SIZE, &cipher_key);
    memcpy(&cipher_key2, record, 12);
    free(record);

    record_key = strtol(argv[1], NULL, 16);
    cipher_key.key1 = strtol(argv[2], NULL, 16);
    cipher_key.key2 = strtol(argv[3], NULL, 16);
    decrypt_block(&cipher_key, &cipher_key2, record_key);

    return 0;
```
void decrypt_block(struct cipher_key *key, struct cipher_key2 *key2, uint16_t sig)
{
    int j, size;
    unsigned char byte;
    unsigned char *cipher;

    printf("Decrypting block with signature: \%x\n", sig);
    cipher = getBlock(bin, &size, sig, FILE_SIZE, key);
    for (j = 0; j < size / 2; j++) {
        uint16_t input;
        memcpy(&input, &cipher[j*2], 2);
        byte = decrypt_byte(key2->key1, key2->key2, key2->key3, input);
        printf("%c", byte);
    }
    printf("\n");
    free(cipher);
}

unsigned char *getBlock (unsigned char *buf, int *recsize, uint16_t match, int length, struct cipher_key *key)
{
    unsigned char header;
    unsigned char *record;
    uint16_t *sig;
    uint32_t i, pos, *size;

    pos = 0;
    while (pos < (length - 5)) {
        header = buf[pos];
        if (header != 0xFD) {
            printf("Invalid header at offset: \%x\n", pos);
            return 0;
        }
        sig = (uint16_t *)&buf[pos+1];
        size = (uint32_t *)&buf[pos+3];
        if (*sig != match) {
            pos += 7;
            pos += *size;
        } else {
            record = malloc(*size);
            *recsize = *size;
            for (i = 0; i < *size; i++) {
                unsigned char *ptr = (unsigned char *) key;
                unsigned char letter = (buf[pos + i + 7] ^ ptr[i % 8]) & 0xFF;
                record[i] = letter;
            }
        }
    }
    printf("%x: No match found\n", match);
    return 0;
}

unsigned char decrypt_byte (uint32_t key1, uint32_t key2, uint32_t key3, uint16_t in)
{
    unsigned char plain;
    int idx, j;
    uint32_t tmp, tmp1, tmp2, tmp3;
    uint64_t big;

    tmp1 = (key1 << 0x10) ^ key1;

if (tmp1 <= 1) {
    tmp1 = key1 << 0x18;
    if (tmp1 <= 1) {
        tmp1 = ~tmp1;
    }
}

tmp2 = (key2 << 0x10) ^ key2;
if (tmp2 <= 7) {
    tmp2 = key2 << 0x18;
    if (tmp2 <= 7) {
        tmp2 = ~tmp2;
    }
}

tmp3 = (key3 << 0x10) ^ key3;
if (tmp3 <= 0xF) {
    tmp3 = key3 << 0x18;
    if (tmp3 <= 1) {
        tmp3 = ~tmp3;
    }
}

for (j = 0; j <= in; j++) {

tmp = ((tmp1 << 0xD) ^ tmp1) >> 0x13;
tmp1 = ((tmp1 & 0xFFFFFFFE) << 0xC) ^ tmp;

tmp = ((tmp2 * 4) ^ tmp2) >> 0x19;
tmp2 = ((tmp2 & 0xFFFFFFF8) << 0x4) ^ tmp;

tmp = ((tmp3 * 8) ^ tmp3) >> 0xB;
tmp3 = ((tmp3 & 0xFFFFFFF0) << 0x11) ^ tmp;
}


tmp1 ^= tmp2;
tmp3 ^= tmp1;
tmp1 = 0x80808081;
tmp2 = tmp3;
big = tmp1;
big ^= tmp2;
tmp2 = big & 0xFFFFFFFF;
tmp1 = big >> 0x20;
tmp1 >>= 0x7;
tmp2 = (tmp1 << 0x8) - tmp1;
tmp3 = tmp2;
plain = tmp3 & 0xFF;

return plain;
## Appendix 3: System Call to Variable Mappings

<table>
<thead>
<tr>
<th>Variable</th>
<th>System Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>dword_61A0</td>
<td>_NSGetExecutablePath</td>
</tr>
<tr>
<td>dword_61A4</td>
<td>CFStringCreateWithCString</td>
</tr>
<tr>
<td>dword_61A8</td>
<td>CFStringGetCString</td>
</tr>
<tr>
<td>dword_61AC</td>
<td>CFRlease</td>
</tr>
<tr>
<td>dword_61B0</td>
<td>CFURLCreateWithString</td>
</tr>
<tr>
<td>dword_61B4</td>
<td>CFHTTPMessageCreateRequest</td>
</tr>
<tr>
<td>dword_61B8</td>
<td>CFHTTPMessageSetHeaderValue</td>
</tr>
<tr>
<td>dword_61BC</td>
<td>CFReadStreamCreateForHTTPRequest</td>
</tr>
<tr>
<td>dword_61C0</td>
<td>CFReadStreamOpen</td>
</tr>
<tr>
<td>dword_61C4</td>
<td>CFReadStreamRead</td>
</tr>
<tr>
<td>dword_61C8</td>
<td>CFReadStreamClose</td>
</tr>
<tr>
<td>dword_61CC</td>
<td>IORegistryEntryWithPath</td>
</tr>
<tr>
<td>dword_61D0</td>
<td>IORegistryEntryCreateCFProperty</td>
</tr>
<tr>
<td>dword_61D4</td>
<td>IObjectRelease</td>
</tr>
<tr>
<td>dword_61D8</td>
<td>uncompress</td>
</tr>
<tr>
<td>dword_61DC</td>
<td>compressBound</td>
</tr>
<tr>
<td>dword_61E0</td>
<td>compress2</td>
</tr>
<tr>
<td>dword_61E4</td>
<td>__CFStringMakeConstantString</td>
</tr>
<tr>
<td>dword_61E8</td>
<td>BIO_new</td>
</tr>
<tr>
<td>dword_61EC</td>
<td>BIO_ctrl</td>
</tr>
<tr>
<td>dword_61F0</td>
<td>BIO_write</td>
</tr>
<tr>
<td>dword_61F4</td>
<td>BIO_free_all</td>
</tr>
<tr>
<td>dword_61F8</td>
<td>BIO_push</td>
</tr>
<tr>
<td>dword_61FC</td>
<td>BIO_new_mem_buf</td>
</tr>
<tr>
<td>dword_6200</td>
<td>BIO_f_base64</td>
</tr>
<tr>
<td>dword_6204</td>
<td>BIO_s_mem</td>
</tr>
<tr>
<td>dword_6208</td>
<td>BIO_read</td>
</tr>
<tr>
<td>dword_620C</td>
<td>RSA_verify</td>
</tr>
<tr>
<td>dword_6210</td>
<td>SHA1</td>
</tr>
<tr>
<td>dword_6214</td>
<td>gethostbyname</td>
</tr>
<tr>
<td>dword_6218</td>
<td>BN_bin2bn</td>
</tr>
<tr>
<td>dword_621C</td>
<td>RSA_new</td>
</tr>
</tbody>
</table>