Bradley Isbell

Chris Anderla

Chuck Gait

Dennis Whichard

Lab 10 - Anti-reversing Techniques

# **Purpose:**

To learn about and apply anti-reversing techniques to existing source code in order to protect a sensitive algorithm.

# **Requirements:**

Windows 7 Professional SP1

Visual Studio Community 2013 Version 12.0.3101.00 Update 4

PEBrowse64 Professional Version 6.0

Kali Linux 1.0.9

IDA Pro 5.0

HxD (or Frhed)

Metasploit Framework (<http://www.metasploit.com/>)

WinDBG 6.3.9600.17298 X86

**Experiment:**

Analysis of SimpleCrackMe key checking algorithm

The program performs a gross check of the given key to eliminate wasting time when the key is obviously not going to work. The function “**doCheckConvert**” is called to do the entire key check including the gross check. The “**user**” and “**key**” strings that were supplied on the command line are passed to the routine. If the gross check passes then the “**doCheck**” routine is called to do a complete check of the key. This gross check consists of a quick check to verify that the given key string consists of 32 characters.

Passing that simple test the routine sequentially takes every two characters of the given key from left to right and treats them as a two digit hex number, converting them from ASCII Hex to an unsigned 8-bit value and putting the result into an unsigned character array starting at 0 and ending at index 15 for the last pair of characters. This array of 16 unsigned bytes and the user string are both passed to the ‘doCheck’ routine for the final verification.

Once processing moves to “**doCheck**” the real work begins. A cryptographic hash is created for SHA1. By definition, a cryptographic hash is “a hash function which is considered practically impossible to invert, that is, it is impossible to recreate the input data from its hash value alone[[1]](#footnote-1)”. A cryptographic hash needs to be:

* easy to compute for any given value;
* impossible to generate a message from its hash;
* impossible to modify the message without changing the hash; and
* no two messages will generate the same hash.

The function first starts out by calling “**CryptAcquireContext”**, giving us a handle to a particular key container. It then calls the hashing API, named “**CryptCreateHash**”, which gives us a handle to our cryptographic service provider hash object; this handle will then be used to hash our data. Next, we send the hash handle to “**CryptHashData**” where we add our data to the hash. The hashing algorithm for this call is utilizing the well-known SHA-1, which has become a standard in today’s cryptography realm. Once the data stream has been processed the function calls the “**CryptGetHashParam**” API to retrieve the 20-byte long array value. If at any time during this process something fails, we release our crypto-context, destroy the hash and return FALSE.

At this point the hashed user string is ready for massaging to get into the format needed for the final compare with the given key once that key is also processed. Both the hashed user string and the given key string need to be processed into a value for comparison.

First the value in “**sha1Data**” is processed by starting with a 0 then taking the value and multiplying it by 31 and then adding the next byte and repeating until this is done for each byte of data. The variable that is used to hold the value being calculated is a 16-bit unsigned value so the result is a 16 bit value. There are two obvious side effects to using a 16 bit value. First, the value calculated can be the same for many different user strings. Second, if you reverse the code and determine this, you know that a brute force attack may have a chance since there are only 64K unique values.

Next we perform a similar operation using the supplied key. This key is processed into a 16 bit unsigned value and will be compared with the value calculated for the user string. In this operation, the value is multiplied by 127 (vs. 31) and then the next byte is added and on it repeats until all 16 bytes have been processed. Here again, the variable holding the value is a 16-bit unsigned.

Then the two values are compared and the results of this comparison is returned to the calling function.

**Experiment:**

First attempt at checking for the presence of a debugger

To get a baseline understanding of the “isdebuggerpresent” function (well known documented API within the Microsoft Windows platform) we decided to write a small function to utilize this functionality and inserted the code into the main program.

Upon startup of the program the “CheckProcessDebugFlag” function is called, and a Boolean value is return based on the presence of the executable running in a debugger. Upon review of the ASM it appeared that simply checking for if a debugger can be easily bypassed utilizing various reverse engineering tools, and a new obfuscation method was tried. The sample experiment can be found in the Appendix A, and was generated based on MSDN at “<https://msdn.microsoft.com/en-us/library/windows/desktop/ms680345(v=vs.85).aspx>”.

When we load SimpleCrackMe.exe in Ollydbg and press CTRL + N we see a list of all the Imports and Exports for the program. In this list we find the function IsDebuggerPresent located in kernel32.dll. Now, to disable the IsDebuggerPresent function, all we have to do is go to the dump window (in Ollydbg), press CTRL+G and go to the appropriate line break. Then we just select the 3rd byte, which will be 01 and change it to 00 (True value to False).

As you can see it was very easy to change the value and bypass the check for a running debugger.

**Experiment:**

**Obfuscated Debugger check and Venom injection**

Sample Code #1:

In this code two functions were added: “**doCheckInputVal**” and “**doCheckHackOutput**”. The names of the functions were chosen so that they would not tip off their true functionality.

The “**doCheckInputVal**” function is an obfuscated version of “**isDebuggerPresent**”. It will return 1 if the program is run in a debugger, it will return 0 if not.

The main function starts by calling “**doCheckInputVal**”, if the return is a ‘1’ then it will call the “**doCheckHashOutput**” function. I have placed a long series of NOP’s in this function. I then used **msfvenom** to generate shell code to open calc.exe. The NOPs were then overwritten with these opcodes for the shell code. The shell code could be much more malicious in nature; **calc.exe** was selected as a proof of concept and to remain non-intrusive. The end result is that if the program is run in a debugger, a calculator will open up and then the program will exit.

// SimpleCrackMe.cpp : Defines the entry point for the console application.

//

#include <Windows.h>

#include <wincrypt.h>

#include <time.h>

#include <stdio.h>

#define DEBUGLINE fprintf(stderr, "DBG: %d\n", \_\_LINE\_\_)

BOOL doCheck(char user[], unsigned char\* key);

int doCheckInputVal(){

unsigned x = 18;

unsigned \*X = &x;

unsigned y = 8;

unsigned \*Y = &y;

unsigned z = 2;

unsigned \*Z = &z;

\_asm {

mov ebx, dword ptr X

mov eax,ebx

mov ebx, [eax]

mov eax, dword ptr Y

add ebx, [eax]

mov eax, dword ptr Z

sub ebx, [eax]

push ebx

pop eax

mov eax,fs:[eax]

mov eax,[eax + 0x30]

mov ebx, dword ptr X

xor ecx,ecx

mov cl,byte ptr[eax + 0x2]

mov dword ptr[ebx], ecx

}

return x;

}

void doCheckHashOutput(){

\_asm{

nop

nop

<...Truncated… 260 nop’s in total>

}

}

BOOL doCheckConvert(char user[], char keychars[]) {

//DEBUGLINE;

if (strlen(keychars) != 32) {

return FALSE;

}

//DEBUGLINE;

unsigned char key[16];

char temp[3] = { 0 };

char\* check;

for (int i = 0; i < 16; i++) {

memcpy(temp, &keychars[2 \* i], 2);

key[i] = strtol(temp, &check, 16);

#ifdef \_DEBUG

fprintf(stderr, "key[%d] = %02hhx\n", i, key[i]);

#endif

if (check != &temp[2]) {

return FALSE;

}

}

//DEBUGLINE;

return doCheck(user, key);

}

BOOL doCheck(char user[], unsigned char\* key) {

HCRYPTPROV hProv = 0;

HCRYPTHASH hHash = 0;

BOOL bResult = FALSE;

bResult = CryptAcquireContext(&hProv, NULL, NULL, PROV\_RSA\_FULL, CRYPT\_VERIFYCONTEXT);

if (!bResult) {

return FALSE;

}

//DEBUGLINE;

bResult = CryptCreateHash(hProv, CALG\_SHA1, 0, 0, &hHash);

if (!bResult) {

CryptReleaseContext(hProv, 0);

return FALSE;

}

//DEBUGLINE;

bResult = CryptHashData(hHash, (const BYTE\*)user, strlen(user), 0);

if (!bResult) {

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

return FALSE;

}

//DEBUGLINE;

BYTE sha1Data[20] = { 0 };

DWORD cbHash = sizeof(sha1Data);

bResult = CryptGetHashParam(hHash, HP\_HASHVAL, sha1Data, &cbHash, 0);

if (!bResult) {

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

return FALSE;

}

//DEBUGLINE;

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

//DEBUGLINE;

#if 0

printf("SHA1(user) = ");

for (int i = 0; i < cbHash; i++) {

printf("%02hhx", sha1Data[i]);

}

printf("\n");

#endif

WORD checkSHA1 = 0;

for (int i = 0; i < cbHash; i++) {

checkSHA1 \*= 31;

checkSHA1 += sha1Data[i];

}

WORD checkKey = 0;

for (int i = 0; i < 16; i++) {

checkKey \*= 127;

checkKey += key[i];

}

#ifdef \_DEBUG

printf("checkSHA1 = %04x, checkKey = %04x\n", checkSHA1, checkKey);

#endif

return checkSHA1 == checkKey;

}

int main(int argc, char\* argv[])

{

int t = doCheckInputVal();

if (t == 1){

doCheckHashOutput();

}

#ifdef \_DEBUG

if (argc == 2) {

unsigned char key[16];

srand(time(NULL));

for (int i = 0; i < 16; i++) {

key[i] = rand();

}

while (1) {

printf("Key: ");

for (int i = 0; i < 16; i++) {

printf("%02hhx", key[i]);

}

printf(": ");

if (doCheck(argv[1], key)) {

break;

}

for (int i = 15; i >= 0; i--) {

key[i]++;

if (key[i] != 0) break;

}

/\*for (int i = 0; i < 16; i++) {

key[i] = rand();

}\*/

}

printf("Found key: ");

for (int i = 0; i < 16; i++) {

printf("%02hhx", key[i]);

}

printf("\n");

goto LAME\_EXIT;

}

#endif

if (argc != 3) {

fprintf(stderr, "Please provide a username and key");

exit(-1);

}

if (doCheckConvert(argv[1], argv[2])) {

printf("You're winner!\n");

}

else {

printf("You lose");

}

LAME\_EXIT:

getc(stdin);

}

# 

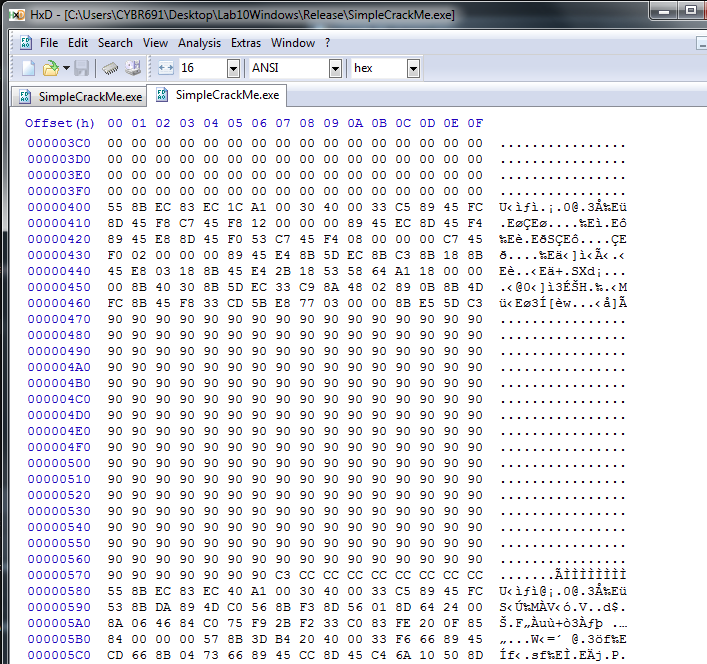
Shellcode to open calc.exe

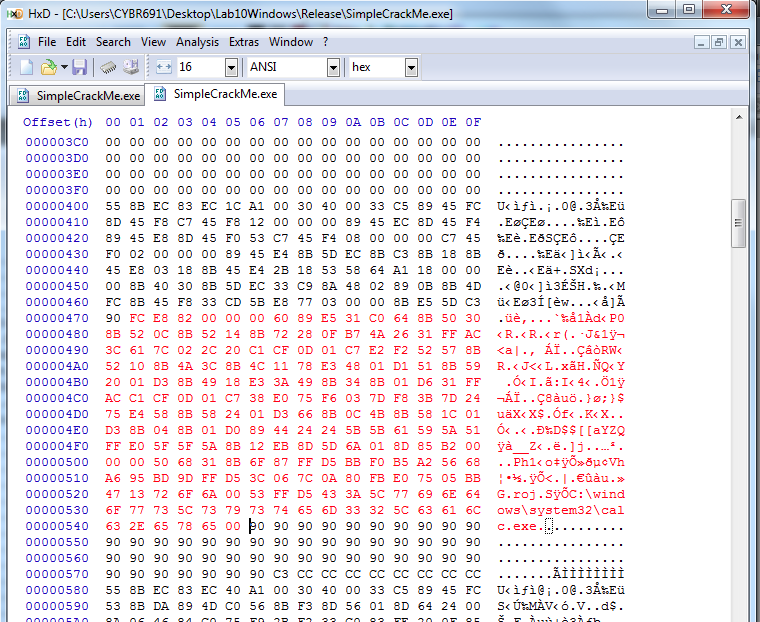
These are the opcodes that were used to overwrite the long series of NOP’s, which would be seen as ‘90’ in the **HxD** hex editor. The following shell code was generated with the msfvenom command which is part of the Metasploit framework.

# msfvenom -p windows/exec CMD=C:\\windows\\system32\\calc.exe -f python | sed -e 's/^buf.\* "//' -e 's/\\x/ /g' -e 's/\"$//'

fc e8 82 00 00 00 60 89 e5 31 c0 64 8b 50 30 8b 52 0c 8b 52 14 8b 72 28 0f b7 4a 26 31 ff ac 3c 61 7c 02 2c 20 c1 cf 0d 01 c7 e2 f2 52 57 8b 52 10 8b 4a 3c 8b 4c 11 78 e3 48 01 d1 51 8b 59 20 01 d3 8b 49 18 e3 3a 49 8b 34 8b 01 d6 31 ff ac c1 cf 0d 01 c7 38 e0 75 f6 03 7d f8 3b 7d 24 75 e4 58 8b 58 24 01 d3 66 8b 0c 4b 8b 58 1c 01 d3 8b 04 8b 01 d0 89 44 24 24 5b 5b 61 59 5a 51 ff e0 5f 5f 5a 8b 12 eb 8d 5d 6a 01 8d 85 b2 00 00 00 50 68 31 8b 6f 87 ff d5 bb f0 b5 a2 56 68 a6 95 bd 9d ff d5 3c 06 7c 0a 80 fb e0 75 05 bb 47 13 72 6f 6a 00 53 ff d5 43 3a 5c 77 69 6e 64 6f 77 73 5c 73 79 73 74 65 6d 33 32 5c 63 61 6c 63 2e 65 78 65 00

Here are the before and after screenshots of replaces the string of NOPs with the shell code.





# 

In the final program define statements were used to instead of a function. One define statement, CHECKDBG, is used to perform the check if a debugger is being used, a second define statement is used to insert the NOP’s that will then be manually modified using HxD. This will make it more difficult to analyze because it will not be seen as a function call, but the code will be in-lined with the rest of the program. The define statements are called two times, once at the beginning of main and once when entering into the getCrypt() function. This way the other team will need to discover and deactivate both occurrences in order to successfully analyze the binary in a debugger.

# 

# **Experiment:**

# **Obfuscate the key verification**

Starting with the original program, we need to do a little something to make it more difficult for the reverser to figure out how to build a key generator. We used most of what the original program did. This includes the encryption of the user string. From there the original took the encrypted string and byte by byte multiplied by 31 then added the next byte, as described earlier. We basically do the same thing but the 31 is changed and we keep saving the result into a 64 bit value to make brute forcing a little more difficult. It is this value that is used to verify if the key is valid for the user name. To do this verification the given key string is converted to a 64 bit value for the comparison. This is where we add a little something to make the key different than the 64 bit value. If we did nothing to it then you could take the 64 bit value, convert it to an ASCII hex string and that would work for a key. One benefit of working with a 64 bit value is that it makes reading 32 bit assembly a little more difficult.

There are also two quick checks on the given key that you will see in the full code listing. First, we limit the size of the key they can give. This is to avoid any overflow problems. Second, we add an additional character at the end of the key. We just go through a few of the first character and grab the largest one and slap it on the end. This provides a method to quickly verify that the key provided is a potential valid key before further processing is performed.

What we do is take the 64 bit value and convert it to an ASCII hex string which will be 16 hex characters. We go through it character by character and we take the integer value of the character and the one after it to get an 8 bit value. If this value is a printable ASCII character, then we use that value in place of the two we looked at. If it isn’t a printable character or if it is a character that can cause problems on the command line (“<” or any quote character, backslash and so on) we don’t use that character. Then we just move to the next character and look at it and the next one. We also need to not convert a pair of character if they happen to have a value that is a valid hex character.

The code to convert the key string into a 64 bit value is in ***buildKeyValueFromStr*** which is listed just below*.* It is really pretty simple so the routine has been sprinkled with bogus variables and math operation to make following it more difficult. The “***then***” portions of each “***if***”could be single line calculations but they have been stretched over multiple lines with the use of a second variable and calculations that cancel out making a simple statement a convoluted mess. There have also been some “**if**” statements added that if true will break out of the loop; the thing is, they will never be true! The ***done*** flag is part of the loop control and it gets modified within the loop but in the end it is always set to false so it is totally useless. Overall it is crackable but it will be a headache for the reverser.

// Given the key string this will build the 64-bit unsigned that corresponds to it.

// Some effort was put into making it things convoluted.

//

unsigned \_\_int64 buildKeyValueFromStr(unsigned char\*keyStr) {

//unsigned sha1Len = sizeof(sha1Data);

unsigned char \*strPtr = keyStr;

unsigned \_\_int64 val = 0;

unsigned \_\_int64 val2 = 0;

unsigned i = 0;

unsigned ch1;

BOOL done = false;

// Some obfuscation was added to make things hard to follow.

while (\*strPtr && !done) {

ch1 = \*strPtr; // Now we have 2 variables that equal the same value, should help obfuscation

if (ch1 < 0x21) {

// val = val \* 16 + unsigned(ch1); // This is what this block is really doing

val2 = ch1 \* \*strPtr + 0x20;

done = true;

val = val \* 16 + (val2 / unsigned(\*strPtr)) - 0x20;

}

else {

if ((\*strPtr > 0x2f) && (ch1 < 0x3a)) { // '0' <= ch1 <= '9'

// val = val \* 16 + ch1 - 0x30; // This is what this block is really doing

val2 = \*strPtr - ' ';

if ((\*strPtr > 0x100) && (\*strPtr <= 0x180)) break; // bogus

val = val \* 16 + ch1 - 0x30;

val += val2;

val2++; // bogus

val = val - ch1 + ' ';

}

else if ((ch1 > 0x60) && (\*strPtr < 0x67)){ // 'a' <= ch1 ,= 'f'

//val = (val \* 16) + unsigned(\*strPtr) - 0x57; // This is what this is really doing

val2 = 0x61;

val = (val \* 16) + unsigned(\*strPtr) - val2;

done = true;

val2 = 10;

val += val2;

}

else {

//val = (val \* 256) + ch1; // This is what this block is really doing

val2 = ch1 + (\* strPtr) \* 4;

val = (val \* 256);

done = false;

val += (val2 / 5);

}

}

done = false;

if ((ch1 < 0) || (\*strPtr > 0x142))

break;

strPtr++;

}

return val;

}

The overall program for this experiment is in Appendix B. This experiment allowed us to play with obfuscating the simple algorithm to make it tougher to reverse. Additionally, the \_DEBUG compiler control was used to allow code to be put into the source file for testing but not show it in the released version. The \_DEBUG control allowed us to have the key generation routine in the source which is much more convenient than creating a separate program. What we implemented was that when only one argument was supplied to the debug version of the program it would take that value generate a key for it then take that key and use it and the given argument and run them through and see if the generated key works for the given string. This comes in handy when debugging the the user/key verification code.

**Results**

The original program was taken and analyzed to determine how it was working. The experiment gives the details that showed us what was there was a pretty simple process.

We ran several experiments to come up with some anti-reversing techniques we used in the final product. Initially we created a procedure that would check for a debugger being present. This version used all C code which was very interesting and worked in detecting the debugger. We built on this experiment and used inline assemble and added additional operations to make the code less obvious. We used this updated debugger check and added it into the original program and when the debugger was detected it would then replace a section of code with the Windows calculator program using MSFVenom to place it into the code. In addition we modified the key generation to take the SHA1 value and process it and convert it into a string. Our program when it checks the user/key pair takes and adds a lot of extraneous code and simple operations are obfuscated to make them difficult to reverse. Our resulting key is more than just a hex-ASCII string, it will include additional printable ASCII characters. All of this was combined into the final program which can be found in Appendix C.

# Appendix A

Added the check to determine if a debugger is present.

// SimpleCrackMe.cpp : Defines the entry point for the console application.

//

#include <Windows.h>

#include <wincrypt.h>

#include <time.h>

#include <stdio.h>

#define DEBUGLINE fprintf(stderr, "DBG: %d\n", \_\_LINE\_\_)

BOOL doCheck(char user[], unsigned char\* key);

BOOL doCheckConvert(char user[], char keychars[]) {

//DEBUGLINE;

if (strlen(keychars) != 32) {

return FALSE;

}

//DEBUGLINE;

unsigned char key[16];

char temp[3] = { 0 };

char\* check;

for (int i = 0; i < 16; i++) {

memcpy(temp, &keychars[2 \* i], 2);

key[i] = strtol(temp, &check, 16);

#ifdef \_DEBUG

fprintf(stderr, "key[%d] = %02hhx\n", i, key[i]);

#endif

if (check != &temp[2]) {

return FALSE;

}

}

//DEBUGLINE;

return doCheck(user, key);

}

BOOL doCheck(char user[], unsigned char\* key) {

HCRYPTPROV hProv = 0;

HCRYPTHASH hHash = 0;

BOOL bResult = FALSE;

bResult = CryptAcquireContext(&hProv, NULL, NULL, PROV\_RSA\_FULL, CRYPT\_VERIFYCONTEXT);

if (!bResult) {

return FALSE;

}

//DEBUGLINE;

bResult = CryptCreateHash(hProv, CALG\_SHA1, 0, 0, &hHash);

if (!bResult) {

CryptReleaseContext(hProv, 0);

return FALSE;

}

//DEBUGLINE;

bResult = CryptHashData(hHash, (const BYTE\*)user, strlen(user), 0);

if (!bResult) {

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

return FALSE;

}

//DEBUGLINE;

BYTE sha1Data[20] = { 0 };

DWORD cbHash = sizeof(sha1Data);

bResult = CryptGetHashParam(hHash, HP\_HASHVAL, sha1Data, &cbHash, 0);

if (!bResult) {

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

return FALSE;

}

//DEBUGLINE;

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

//DEBUGLINE;

#if 0

printf("SHA1(user) = ");

for (int i = 0; i < cbHash; i++) {

printf("%02hhx", sha1Data[i]);

}

printf("\n");

#endif

WORD checkSHA1 = 0;

for (int i = 0; i < cbHash; i++) {

checkSHA1 \*= 31;

checkSHA1 += sha1Data[i];

}

WORD checkKey = 0;

for (int i = 0; i < 16; i++) {

checkKey \*= 127;

checkKey += key[i];

}

#ifdef \_DEBUG

printf("checkSHA1 = %04x, checkKey = %04x\n", checkSHA1, checkKey);

#endif

return checkSHA1 == checkKey;

}

**//Code Added To Check For Debugger - Called In Main**

inline bool CheckProcessDebugFlags()

{

typedef NTSTATUS(WINAPI \*pNtQueryInformationProcess)

(HANDLE, UINT, PVOID, ULONG, PULONG);

DWORD NoDebugInherit = 0;

NTSTATUS Status;

pNtQueryInformationProcess NtQIP = (pNtQueryInformationProcess)

GetProcAddress(GetModuleHandle(TEXT("ntdll.dll")),

"NtQueryInformationProcess");

Status = NtQIP(GetCurrentProcess(),

0x1f,

&NoDebugInherit, 4, NULL);

if (Status != 0x00000000)

return false;

if (NoDebugInherit == FALSE)

return true;

else

return false;

}

int main(int argc, char\* argv[])

{

#ifdef \_DEBUG

if (argc == 2) {

unsigned char key[16];

srand(time(NULL));

for (int i = 0; i < 16; i++) {

key[i] = rand();

}

while (1) {

printf("Key: ");

for (int i = 0; i < 16; i++) {

printf("%02hhx", key[i]);

}

printf(": ");

if (doCheck(argv[1], key)) {

break;

}

for (int i = 15; i >= 0; i--) {

key[i]++;

if (key[i] != 0) break;

}

/\*for (int i = 0; i < 16; i++) {

key[i] = rand();

}\*/

}

printf("Found key: ");

for (int i = 0; i < 16; i++) {

printf("%02hhx", key[i]);

}

printf("\n");

goto LAME\_EXIT;

}

#endif

if (argc != 3) {

fprintf(stderr, "Please provide a username and key");

exit(-1);

}

**//Call To Check For Debugger, And If Present Sleep 20 Seconds And Exit**

if (CheckProcessDebugFlags()) {

Sleep(20000);

exit(0);

}

if (doCheckConvert(argv[1], argv[2])) {

printf("You're winner!\n");

}

else {

printf("You lose");

}

LAME\_EXIT:

getc(stdin);

}

# Appendix B

This is the complete listing of the new SimpleCrackMe.cpp with just the code obfuscation.

// SimpleCrackMe.cpp : Defines the entry point for the console application.

//

#include <Windows.h>

#include <wincrypt.h>

#include <time.h>

#include <stdio.h>

#define DEBUGLINE fprintf(stderr, "DBG: %d\n", \_\_LINE\_\_)

const unsigned \_\_int64 MASH\_FACTOR = 33;

// This is from the original program it just uses a const for the value

// which was changed from 31 to something else.

//

unsigned \_\_int64 buildKeyValueFromSHA1(BYTE \*sha1Data, int sha1Len) {

//unsigned sha1Len = sizeof(sha1Data);

unsigned \_\_int64 val = 0;

for (int i = 0; i < sha1Len; i++) {

val \*= MASH\_FACTOR;

val += sha1Data[i];

}

return val;

}

// Just converts a value to its ASCII-hex value. inlined to make things messy where used.

//

inline char hexDigit(unsigned n)

{

if (n < 0x3a) {

return n - 0x30;

}

if (n < 0x47)

return (n - 0x41 + 10);

else {

return (n - 0x61 + 10);

}

}

// Given the key string this will build the 64-bit unsigned that corresponds to it.

// Some effort was put into making it things convoluted.

//

unsigned \_\_int64 buildKeyValueFromStr(unsigned char\*keyStr) {

//unsigned sha1Len = sizeof(sha1Data);

unsigned char \*strPtr = keyStr;

unsigned \_\_int64 val = 0;

unsigned \_\_int64 val2 = 0;

unsigned i = 0;

unsigned ch1;

BOOL done = false;

// Some obfuscation was added to make things hard to follow.

while (\*strPtr && !done) {

ch1 = \*strPtr; // Now we have 2 variables that equal the same value, should help obfuscation

if (ch1 < 0x21) {

// val = val \* 16 + unsigned(ch1); // This is what this block is really doing

val2 = ch1 \* \*strPtr + 0x20;

done = true;

val = val \* 16 + (val2 / unsigned(\*strPtr)) - 0x20;

}

else {

if ((\*strPtr > 0x2f) && (ch1 < 0x3a)) { // '0' <= ch1 <= '9'

// val = val \* 16 + ch1 - 0x30; // This is what this block is really doing

val2 = \*strPtr - ' ';

if ((\*strPtr > 0x100) && (\*strPtr <= 0x180)) break; // bogus

val = val \* 16 + ch1 - 0x30;

val += val2;

val2++; // bogus

val = val - ch1 + ' ';

}

else if ((ch1 > 0x60) && (\*strPtr < 0x67)){ // 'a' <= ch1 ,= 'f'

//val = (val \* 16) + unsigned(\*strPtr) - 0x57; // This is what this is really doing

val2 = 0x61;

val = (val \* 16) + unsigned(\*strPtr) - val2;

done = true;

val2 = 10;

val += val2;

}

else {

//val = (val \* 256) + ch1; // This is what this block is really doing

val2 = ch1 + (\* strPtr) \* 4;

val = (val \* 256);

done = false;

val += (val2 / 5);

}

}

done = false;

if ((ch1 < 0) || (\*strPtr > 0x142))

break;

strPtr++;

}

#ifdef \_DEBUGnot

printf("buildKeyValueFromStr: Converted %s key string to value %16I64x\n", keyStr, val);

#endif

return val;

}

// Not called but could be an alternate to the terminater char

//

unsigned short buildCheckSum(char \*str)

{

unsigned short val = 0;

for (int i = 0; i < strlen(str); i++)

val += (unsigned short)str[i];

return val;

}

// Just looks over the second through the 8th chars and finds the biggest one

//

char findTerminator(char \*key)

{

char ch = 0x21;

for (int i = 1; i < 7; i++)

if (key[i] > ch)

ch = key[i];

else if (key[i] <= 0x20)

break;

return ch;

}

#ifdef \_DEBUG

// Takes the 64-bit unsigned value generated by getCrypt and adds our twist.

// This is spaghetti mess. Lots of poor coding to make it difficult to reverse.

// Its hard enough to follow with the source.

// Basically it goes through the ASCII-hex version of the key and if any the consecutive chars

// is an ASCII printable char that is put into the string in their place. There are a bunch of

// chars that are not used because it causes problems on the command line.

// And now I realize this is not going to be in the release so I didn't need to convolute it!

void buildKeyStrFromValue(unsigned \_\_int64 keyVal, char \*outStr) {

char str[64];

int val;

// Convert the value into a hex string

sprintf\_s(str, "%16I64x", keyVal);

BOOL done = false;

unsigned i = 0;

int j = 0;

unsigned ch1, ch2;

int len = strlen(str);

val = 17; // Psyche

while (!done) {

ch1 = str[i];

outStr[j] = 0x33; // Psyche, useless

if (i + 1 < len)

ch2 = str[i + 1];

else {

outStr[j] = ch1;

done = false; // Psyche

j++;

break;

}

val = hexDigit(ch1) \* 16;

val += hexDigit(ch2);

outStr[j] = val;

if (val > 0x7e) {

outStr[j] = ch1;

j++;

done = true; // Psyche

}

else {

if (val > 0x20)

outStr[j] = val;

if ((val > 0x2f) && (val < 0x3a) || (val < 0x21)) {

outStr[j] = ch1;

done = false; // Psyche

j = j + 1;

}

// This excludes special chars that cause problem if used on the command line as part of a arg

else if ((val == 0x22) || (val == 0x27) || (val == 0x2c) || (val == 0x7C) || (val == 0x5c) || (val == 0x60) || (val == 0x26) || (val == 0x3e) || (val == 0x3c) || (val == 0x5e) || (val == 0x25)) {

outStr[j] = ch1;

j++;

}

else if ((val > 0x60) && (val < 0x67)) {

outStr[j] = ch1;

done = true; // Psyche

j = j + 1;

}

else if (val < 0x21)

outStr[j++] = ch1;

else {

outStr[j++] = val;

i++;

}

}

done = false; // Psyche

i++;

}

outStr[j] = 0;

}

#endif

// Does all the encryption work on the user name then converts it into a 64-bit unsigned

//

unsigned \_\_int64 getCrypt(char user[]) {

HCRYPTPROV hProv = 0;

HCRYPTHASH hHash = 0;

BOOL bResult = FALSE;

bResult = CryptAcquireContext(&hProv, NULL, NULL, PROV\_RSA\_FULL, CRYPT\_VERIFYCONTEXT);

if (!bResult) {

return FALSE;

}

//DEBUGLINE;

bResult = CryptCreateHash(hProv, CALG\_SHA1, 0, 0, &hHash);

if (!bResult) {

CryptReleaseContext(hProv, 0);

return FALSE;

}

//DEBUGLINE;

bResult = CryptHashData(hHash, (const BYTE\*)user, strlen(user), 0);

if (!bResult) {

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

return FALSE;

}

//DEBUGLINE;

BYTE sha1Data[20] = { 0 };

DWORD cbHash = sizeof(sha1Data);

bResult = CryptGetHashParam(hHash, HP\_HASHVAL, sha1Data, &cbHash, 0);

if (!bResult) {

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

return FALSE;

}

//DEBUGLINE;

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

//DEBUGLINE;

unsigned \_\_int64 ourKeyVal = buildKeyValueFromSHA1(sha1Data, sizeof(sha1Data));

return ourKeyVal;

}

// Gathers the info for the key validation

//

BOOL doCheck(char user[], unsigned char\* key) {

// Encrypt user with SHA! then convert it into our own ulong long

unsigned \_\_int64 ourKeyVal = getCrypt(user);

// Take the given key string and convert it to our own ulong long

unsigned \_\_int64 givenKeyVal = buildKeyValueFromStr(key);

BOOL result1 = (ourKeyVal == givenKeyVal);

// Check against the generated key also? but then we have to put the key gen code in

return result1;

}

// Does a quick sanity check then passes the info for validation

//

BOOL doCheckConvert(char user[], char \*keychars) {

//DEBUGLINE;

// As a quick check we verify the trailer char is correct

char terminatorChar = findTerminator(keychars);

if (keychars[strlen(keychars)-1] != terminatorChar)

return false;

// Remove the trailer char that we added for good measure

keychars[strlen(keychars)-1] = '\0';

//DEBUGLINE;

// Does the all the work and returns the status

return doCheck(user, (unsigned char\*)keychars);

}

#ifdef \_DEBUG

// Given a string this will generate the key for it

//

char \*genKey(char \*user, char \*str)

{

char temp[1024];

strcpy\_s(temp, user);

//printf("genKey: Generateing key for %s\n", temp);

unsigned \_\_int64 ourKeyVal = getCrypt(temp);

buildKeyStrFromValue(ourKeyVal, str);

char terminatorChar = findTerminator(str);

str[strlen(str) + 1] = 0x00;

str[strlen(str)] = terminatorChar;

return str;

}

#endif

//\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\//

int main(int argc, char\* argv[])

{

#ifdef \_DEBUG

// This will test the program. It will take in the user name gen a key then try the key.

// This will help to try many user names and make sure the conversions back and forth work.

//

if (argc == 2){

// If the user name it huge then have them try again

if (strlen(argv[1]) > 64) {

printf("The user name is ridiculous, try again\n");

exit(-1);

}

// Generate the key

char str[100] = { 0 };

genKey(argv[1], str);

printf("The key for: %s is %s\n", argv[1], str);

printf("Self testing %s with generated key %s\n", argv[1], str);

// Use the key to see if is reported to be valid

if (doCheckConvert(argv[1], str)) {

printf("Worked!\n");

}

else {

printf("!!! Failed, got a key/verification problem !!!\n");

}

exit (0);

}

#endif

// Here is the normal path, the given key will be checked to be valid for the user

if (argc != 3) {

fprintf(stderr, "Please provide a username and key\n");

exit(-1);

}

if (doCheckConvert(argv[1], argv[2])) {

printf("You're winner!\n");

}

else {

printf("You lose\n");

}

getc(stdin);

}

# Appendix C

This is the complete listing of the new SimpleCrackMe.cpp with the code obfuscation and the debugger checking code.

// SimpleCrackMe.cpp : Defines the entry point for the console application.

//

#include <Windows.h>

#include <wincrypt.h>

#include <time.h>

#include <stdio.h>

#define DEBUGLINE fprintf(stderr, "DBG: %d\n", \_\_LINE\_\_)

#define CHECKDBG()\

unsigned x = 18;\

unsigned \*X = &x;\

unsigned y = 8;\

unsigned \*Y = &y;\

unsigned z = 2;\

unsigned \*Z = &z;\

\_asm {mov ebx, dword ptr X};\

\_asm {mov eax, ebx};\

\_asm {mov ebx, [eax]};\

\_asm {mov eax, dword ptr Y};\

\_asm {add ebx, [eax]};\

\_asm {mov eax, dword ptr Z};\

\_asm {sub ebx, [eax]};\

\_asm {push ebx};\

\_asm {pop eax};\

\_asm {mov eax, fs:[eax]};\

\_asm {add ebx, ebx};\

\_asm {add ebx, eax};\

\_asm {mov eax, [ebx]};\

\_asm {mov ebx, dword ptr X};\

\_asm {xor ecx, ecx};\

\_asm {mov cl, byte ptr[eax + 0x2]};\

\_asm {mov dword ptr[ebx], ecx};\

#define NOPSLED()\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

\_asm{nop};\

const unsigned \_\_int64 MASH\_FACTOR = 33;

// This is from the original program it just uses a const for the value

// which was changed from 31 to something else.

//

unsigned \_\_int64 buildKeyValueFromSHA1(BYTE \*sha1Data, int sha1Len) {

//unsigned sha1Len = sizeof(sha1Data);

unsigned \_\_int64 val = 0;

for (int i = 0; i < sha1Len; i++) {

val \*= MASH\_FACTOR;

val += sha1Data[i];

}

return val;

}

// Just converts a value to its ASCII-hex value. inlined to make things messy where used.

//

inline char hexDigit(unsigned n)

{

if (n < 0x3a) {

return n - 0x30;

}

if (n < 0x47)

return (n - 0x41 + 10);

else {

return (n - 0x61 + 10);

}

}

// Given the key string this will build the 64-bit unsigned that corresponds to it.

// Some effort was put into making it things convoluted.

//

unsigned \_\_int64 buildKeyValueFromStr(unsigned char\*keyStr) {

//unsigned sha1Len = sizeof(sha1Data);

unsigned char \*strPtr = keyStr;

unsigned \_\_int64 val = 0;

unsigned \_\_int64 val2 = 0;

unsigned i = 0;

unsigned ch1;

BOOL done = false;

// Some obfuscation was added to make things hard to follow.

while (\*strPtr && !done) {

ch1 = \*strPtr; // Now we have 2 variables that equal the same value, should help obfuscation

if (ch1 < 0x21) {

// val = val \* 16 + unsigned(ch1); // This is what this block is really doing

val2 = ch1 \* \*strPtr + 0x20;

done = true;

val = val \* 16 + (val2 / unsigned(\*strPtr)) - 0x20;

}

else {

if ((\*strPtr > 0x2f) && (ch1 < 0x3a)) { // '0' <= ch1 <= '9'

// val = val \* 16 + ch1 - 0x30; // This is what this block is really doing

val2 = \*strPtr - ' ';

if ((\*strPtr > 0x100) && (\*strPtr <= 0x180)) break; // bogus

val = val \* 16 + ch1 - 0x30;

val += val2;

val2++; // bogus

val = val - ch1 + ' ';

}

else if ((ch1 > 0x60) && (\*strPtr < 0x67)){ // 'a' <= ch1 ,= 'f'

//val = (val \* 16) + unsigned(\*strPtr) - 0x57; // This is what this is really doing

val2 = 0x61;

val = (val \* 16) + unsigned(\*strPtr) - val2;

done = true;

val2 = 10;

val += val2;

}

else {

//val = (val \* 256) + ch1; // This is what this block is really doing

val2 = ch1 + (\*strPtr) \* 4;

val = (val \* 256);

done = false;

val += (val2 / 5);

}

}

done = false;

if ((ch1 < 0) || (\*strPtr > 0x142))

break;

strPtr++;

}

#ifdef \_DEBUGnot

printf("buildKeyValueFromStr: Converted %s key string to value %16I64x\n", keyStr, val);

#endif

return val;

}

// Not called but could be an alternate to the terminater char

//

unsigned short buildCheckSum(char \*str)

{

unsigned short val = 0;

for (int i = 0; i < strlen(str); i++)

val += (unsigned short)str[i];

return val;

}

// Just looks over the second through the 8th chars and finds the biggest one

//

char findTerminator(char \*key)

{

char ch = 0x21;

for (int i = 1; i < 7; i++)

if (key[i] > ch)

ch = key[i];

else if (key[i] <= 0x20)

break;

return ch;

}

#ifdef \_DEBUG

// Takes the 64-bit unsigned value generated by getCrypt and adds our twist.

// This is spaghetti mess. Lots of poor coding to make it difficult to reverse.

// Its hard enough to follow with the source.

// Basically it goes through the ASCII-hex version of the key and if any the consecutive chars

// is an ASCII printable char that is put into the string in their place. There are a bunch of

// chars that are not used because it causes problems on the command line.

// And now I realize this is not going to be in the release so I didn't need to convolute it!

void buildKeyStrFromValue(unsigned \_\_int64 keyVal, char \*outStr) {

char str[64];

int val;

// Convert the value into a hex string

sprintf\_s(str, "%16I64x", keyVal);

BOOL done = false;

unsigned i = 0;

int j = 0;

unsigned ch1, ch2;

int len = strlen(str);

val = 17; // Psyche

while (!done) {

ch1 = str[i];

outStr[j] = 0x33; // Psyche, useless

if (i + 1 < len)

ch2 = str[i + 1];

else {

outStr[j] = ch1;

done = false; // Psyche

j++;

break;

}

val = hexDigit(ch1) \* 16;

val += hexDigit(ch2);

outStr[j] = val;

if (val > 0x7e) {

outStr[j] = ch1;

j++;

done = true; // Psyche

}

else {

if (val > 0x20)

outStr[j] = val;

if ((val > 0x2f) && (val < 0x3a) || (val < 0x21)) {

outStr[j] = ch1;

done = false; // Psyche

j = j + 1;

}

// This excludes special chars that cause problem if used on the command line as part of a arg

else if ((val == 0x22) || (val == 0x27) || (val == 0x2c) || (val == 0x7C) || (val == 0x5c) || (val == 0x60) || (val == 0x26) || (val == 0x3e) || (val == 0x3c) || (val == 0x5e) || (val == 0x25)) {

outStr[j] = ch1;

j++;

}

else if ((val > 0x60) && (val < 0x67)) {

outStr[j] = ch1;

done = true; // Psyche

j = j + 1;

}

else if (val < 0x21)

outStr[j++] = ch1;

else {

outStr[j++] = val;

i++;

}

}

done = false; // Psyche

i++;

}

outStr[j] = 0;

}

#endif

// Does all the encryption work on the user name then converts it into a 64-bit unsigned

//

unsigned \_\_int64 getCrypt(char user[]) {

HCRYPTPROV hProv = 0;

HCRYPTHASH hHash = 0;

BOOL bResult = FALSE;

CHECKDBG();

if (x != 1) { goto S; };

C: NOPSLED();

goto C;

S: \_asm{nop};

bResult = CryptAcquireContext(&hProv, NULL, NULL, PROV\_RSA\_FULL, CRYPT\_VERIFYCONTEXT);

if (!bResult) {

return FALSE;

}

//DEBUGLINE;

bResult = CryptCreateHash(hProv, CALG\_SHA1, 0, 0, &hHash);

if (!bResult) {

CryptReleaseContext(hProv, 0);

return FALSE;

}

//DEBUGLINE;

bResult = CryptHashData(hHash, (const BYTE\*)user, strlen(user), 0);

if (!bResult) {

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

return FALSE;

}

//DEBUGLINE;

BYTE sha1Data[20] = { 0 };

DWORD cbHash = sizeof(sha1Data);

bResult = CryptGetHashParam(hHash, HP\_HASHVAL, sha1Data, &cbHash, 0);

if (!bResult) {

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

return FALSE;

}

//DEBUGLINE;

CryptReleaseContext(hProv, 0);

CryptDestroyHash(hHash);

//DEBUGLINE;

unsigned \_\_int64 ourKeyVal = buildKeyValueFromSHA1(sha1Data, sizeof(sha1Data));

return ourKeyVal;

}

// Gathers the info for the key validation

//

BOOL doCheck(char user[], unsigned char\* key) {

// Encrypt user with SHA! then convert it into our own ulong long

unsigned \_\_int64 ourKeyVal = getCrypt(user);

// Take the given key string and convert it to our own ulong long

unsigned \_\_int64 givenKeyVal = buildKeyValueFromStr(key);

BOOL result1 = (ourKeyVal == givenKeyVal);

// Check against the generated key also? but then we have to put the key gen code in

return result1;

}

// Does a quick sanity check then passes the info for validation

//

BOOL doCheckConvert(char user[], char \*keychars) {

//DEBUGLINE;

// As a quick check we verify the trailer char is correct

char terminatorChar = findTerminator(keychars);

if (keychars[strlen(keychars) - 1] != terminatorChar)

return false;

// Remove the trailer char that we added for good measure

keychars[strlen(keychars) - 1] = '\0';

//DEBUGLINE;

// Does the all the work and returns the status

return doCheck(user, (unsigned char\*)keychars);

}

#ifdef \_DEBUG

// Given a string this will generate the key for it

//

char \*genKey(char \*user, char \*str)

{

char temp[1024];

strcpy\_s(temp, user);

//printf("genKey: Generateing key for %s\n", temp);

unsigned \_\_int64 ourKeyVal = getCrypt(temp);

buildKeyStrFromValue(ourKeyVal, str);

char terminatorChar = findTerminator(str);

str[strlen(str) + 1] = 0x00;

str[strlen(str)] = terminatorChar;

return str;

}

#endif

//\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\//

int main(int argc, char\* argv[])

{

CHECKDBG();

if (x != 1) { goto R; };

NOPSLED();

R: \_asm{nop};

#ifdef \_DEBUG

// This will test the program. It will take in the user name gen a key then try the key.

// This will help to try many user names and make sure the conversions back and forth work.

//

if (argc == 2){

// If the user name it huge then have them try again

if (strlen(argv[1]) > 64) {

printf("The user name is ridiculous, try again\n");

exit(-1);

}

// Generate the key

char str[100] = { 0 };

genKey(argv[1], str);

printf("The key for: %s is %s\n", argv[1], str);

printf("Self testing %s with generated key %s\n", argv[1], str);

// Use the key to see if is reported to be valid

if (doCheckConvert(argv[1], str)) {

printf("Worked!\n");

}

else {

printf("!!! Failed, got a key/verification problem !!!\n");

}

exit(0);

}

#endif

// Here is the normal path, the given key will be checked to be valid for the user

if (argc != 3) {

fprintf(stderr, "Please provide a username and key\n");

exit(-1);

}

if (doCheckConvert(argv[1], argv[2])) {

printf("You're winner!\n");

}

else {

printf("You lose\n");

}

getc(stdin);

}

1. Wikipedia, http://en.wikipedia.org/wiki/Cryptographic\_hash\_function. Accessed 24 April 2015. [↑](#footnote-ref-1)