

## Intro to Reverse Engineering - Part 2

In Part 1, Intro to Reverse Engineering - No Assembly Required, we extended the series of coding articles for non-programmers with an area of high interest in the infosec community. We're proud to be able to bring you the highly anticipated follow-up complete with screen shots, sample code and applications. This one is long and detailed, so strap yourselves in for some great educational content.

This paper is designed to outline some essential reverse engineering concepts, tools and techniques - primarily, debuggers and using the debugging process to reverse engineer application functions and algorithms. It is assumed you have knowledge of basic assembly and C programming. An understanding of Win32 programming and API calls is also helpful. This tutorial does not necessarily have to be read in order (although it is strongly advised), as some sections do not contain information that directly relates to subsequent sections. However, if you begin skipping around and find that you have trouble understanding a concept, or feel like you missed an explanation, it would be best to go back to previous sections of the tutorial and read them first.

del.icio.us

Discuss in Forums {mos\_smf\_discuss:Heffner}

Before we begin...

The Windows platform was chosen as the focus of this paper. Because of Window's wide-spread use and its closed-source nature, RCE is performed most often on Windows applications. This tutorial is time-consuming and intense, so let's do a quick overview of the topics that will be covered:

- Introduction to debuggers, specifically, OllyDbg.
- Finding the beginning of the actual code (i.e., the main() function as opposed to code generated automatically by the compiler).
- Identifying how source code is translated into assembly instructions.
- Locating specific functions and API calls.
- Identifying and un-packing basic packers.
- Real-time debugging and patching.

- Serial fishing.
- Self-key generation.
- Translating disassembled functions and algorithms into higher-level languages (aka, creating key-generation programs the right way).
- Identifying and exploiting stack-based buffer overflows in closed source programs.

## Helpful Resources

There are two directories inside the tutorial's zip file: apps, and source. The apps folder contains the pre-compiled programs we will be working with, and the source folder contains the applicable source code for all programs which I have written for this tutorial.

Several tools will be used throughout this tutorial. While it is helpful to have them all, it is really only necessary to have OllyDbg, as you are expected to be tracing through the example programs as they are being presented in the text; this will facilitate your understanding of the text and code.

1. Olly Debugger - For debugging and disassembly.

2. OllyDump - Plugin for Olly.

3. PeID - For identifying packers/encryptors.

4. Dev-C++ - For compiling source with GCC.

5. ActiveState Perl - For creating some exploit code.

## Index

- A 30-Second Intro to OllyDbg - Getting to Know Your Debugger

A quick one page introduction to the basic commands for using Olly.

- Finding Main() - Compiler Code vs Developer Code

Here we look at some different ways to find the main() function, and also examine the differences between program entry points when they are compiled by different compilers.

- EXE Patching

How to temporarily modify assembly instructions in Olly, as well as saving them permanently.

- Intro to Keygens and Un-packing

Learn some basics about executable packers, do some more patching, and create our first keygen.

- Reversing Key Generation Algorithms - Writing a Real Key Generator

Here we completely reverse a key-validation algorithm, and write a full-fledged key generator in C.

- Discovering and Exploiting Buffer Overflows

In the final section of this tutorial, we analyze a program for BOF vulnerabilities, and write some exploit code that allows

us to take control of the EIP.

## A 30-Second Intro to OllyDbg - Getting to Know Your Debugger

Knowing what tools to use, and how to use them, is critical when reversing software. The tool that will most commonly be used throughout this paper is OllyDbg, a popular Win32 debugger. Olly has many plug-ins and advanced features, but we will learn just the essential commands here.

For those unfamiliar, a debugger allows you to view the assembly instructions of a program as they execute: you can set breakpoints on specific instructions to pause execution, single step through the program one instruction at a time, or just let the program run normally. To get a feel for this, open up test.exe in Olly:

The highlighted instruction ('PUSH EBP') is the entry point of the program. There are four main windows in Olly: the code window in the center, the register window on the right, the stack window in the bottom right hand corner and the memory dump window in the bottom left corner.

Instructions and data in the code and memory dump windows can be manipulated by highlighting the desired data and pressing the space bar. Stack and register contents can be modified by right clicking them and selecting 'Modify'. To single step one instruction at a time, press the F8 key: you will see that the highlighted line goes to the next instruction, 'MOV EBP,ESP'. If you want to step into a function call, press the F7 key. The F2 key allows you to set a breakpoint on an instruction, and F9 runs the program normally (still 'inside' of Olly though); to pause the program, press F12. The use of these functions will become clearer as we work through some examples; for now, play around with them, and make note of how each instruction manipulates the stack, registers, and memory.

## Finding Main() - Compiler Code vs Developer Code

If you look at the code at the EP of test.exe and compare it with the source code in test.c, you'll notice that they look nothing alike. The code immediately after the EP has several function calls to APIs like `__set_app_type` and `atexit`. That is because when source code is compiled, the compiler adds some code to the beginning of the program in order to properly initialize the application before executing the code written by the programmer. Different compilers will do drastically different things, so you can usually identify what compiler was used as soon as you see the EP. This particular program was compiled with Dev-C++ which uses the GCC compiler; however, in the vast majority of cases, you don't care much about the compiler code - you want to see what the programmer wrote. So, let's see how we can identify when the switch from compiler code to developer code is made.

Press F8 a several times to single step through the first few instructions, and you will notice that after you execute the 'CALL test.00401100' instruction, the program runs normally. That is because the main program loop (or a call leading to the main program loop) is located at 0041100. Because we pressed F8 to execute the call instruction, Olly executed everything inside that call instruction, and because our program waits for user input before returning from the main loop, the call instruction has not completed yet. To remedy this, restart the program in Olly (Ctrl+F2), press F8 until the next instruction to be executed is 'CALL test.00401100'; this time, press F7 to step into the instruction, and you will be at the first instruction of the function at 00401100:

Let's take a moment to examine this function before executing anything...calls to SetUnhandledExceptionFilter and \_\_getmainargs? This still doesn't look like our main function, so the main function is probably called from inside of this one. Scroll down in the code window until you see this code:

```
CALL test.00401290
```

```
MOV EBX,EAX
```

```
CALL <JMP.&msvcrt._cexit>
```

```
MOV DWORD PTR SS:[ESP],EBX
```

```
CALL <JMP.&KERNEL32.ExitProcess>
```

Notice there is a call to `_cexit` followed by a jump to `ExitProcess`. Well, if `ExitProcess` is being called, then the call to our main function has to occur before this. Most of the function calls are located in `msvcrt.dll` (the '&msvcrt' before the function name is a dead give away), but look at the call just before the call to `_cexit` ('CALL test.00401290'). It is referenced by a memory address rather than an API function name. Select the 'CALL test.00401290' instruction and press F2 to set a breakpoint on it. Now, press F9 to let the program run normally; when the code execution reaches the breakpoint the application will pause on it before executing the instruction. When Olly breaks there, press F7 to step into the call. We are now at memory address 00401290, and the function here looks like our main loop - one call to `printf()`, followed by one call to `getchar()`, followed by a return:

In general, the compiler code will call the main loop just before a call to `_cexit/ExitProcess`, so looking for those two functions can help in finding the start of the main function. The Visual C++ compiler code is a bit more complicated, so we will take a look at how that differs from GCC, but first let's examine our main loop while we're here. There is a lot of data moving in and out of EAX just before the call to `printf()`, so let's see if we can figure out what's going on.

Press F8 until you arrive at the instruction 'MOV EAX, DWORD PTR DS:[403000]'. This instruction is moving the contents located at the memory address of 00403000 into the EAX register. The very next instruction moves the contents of EAX into a local variable on the stack starting 28 bytes below EBP. Well, let's see what is located at 00403000: select the memory dump window in the lower left hand side and press 'Ctl+G'. Enter '403000' (no quotes) at in the text box and click OK. Starting at address 00403000, we see our text string that is displayed by the printf() function:

Since the EAX register can only hold four bytes at a time, only the first four bytes of our string have been copied into the local variable; the subsequent instructions move the rest of the string into the variable's address space four bytes at a time. Finally, the address of the local variable is loaded into EAX, and EAX and the "%s" string are placed on the stack as arguments to the printf() function.

Why did the program go through all the trouble of copying the string in four byte increments onto the stack, when it could have just referenced the original memory address where the string was located (00403000)? Well, because we told it to. In the source code, test.c, you can see that we declared a variable named 'string', and set it equal to the string we wanted passed to printf(). Since a 32-bit processor can only move 32 bits at a time, it had to copy the string from the data section of memory into the string variable four bytes at a time. If we had made the string variable a pointer, the program would have only needed to copy the 4 byte memory address of where the string was located into the variable (the pointer would then point to address 00403000). This would have required only two mov instructions rather than sixteen; it also would have eliminated the need for the LEA instruction. That's almost a 200% decrease in the number of instructions contained in our main function. Granted, this is a very simple program and CPUs are very fast these days, so no measurable difference is seen by the end user. But it is a good example of how disassembly can be used to further our knowledge of high-level programming and help us to develop more efficient applications.

I mentioned earlier that different compilers will produce different assembly instructions for the same source code. Since VC++ is the most commonly used compiler for Windows applications, I've compiled the code in test.c using VC++, and saved it as test\_vc.exe. Opening test\_vc.exe in Olly reveals a vastly different set of assembly instructions that we saw in test.exe:

You may have been wondering why you received a warning message from Olly that test\_vc.exe had an entry point outside of the code section. Without delving too much into the subject of PE (Portable Executable) headers, each Windows program has a PE header which, among other things, identifies which sections of the program contain the code, which sections contain data, etc.

Here, we see that our EP is actually a jump to 00411840, but before we go there, let's take a look at the surrounding jumps. There are a lot of jumps to a lot of different functions, including printf()...right-click the code window and select 'Search for -> All intermodular calls'. This brings up a window that shows all of the function calls in test\_vc.exe that are external to the program (i.e., API functions). Comparing the list of functions here with the list of jumps in the code window, we see that there is a jump for nearly every inter-modular function, as well as several functions that are listed only by address (these are internal to the program). The way this works is, whenever a function call is made within the

program, instead of calling the memory address where the actual function is located, it calls the memory address where the jump to the function is located. For example, the jump to `printf()` is located at `0041118B`, so wherever there is a call to `printf()` within the program, it will actually call `0041118B`, which in turn jumps to the actual location of the `printf()` function. The advantage of this is that if you have a function that is called from multiple locations in your code, and the memory address of that function changes, you only have to change where the jump points to, rather than every single call to that function. It should be noted that depending on compiler options that have been set, the EP code may differ, and in fact, you will normally not see an EP that looks like this when debugging a real-world application; however, this was compiled using the default settings for VC++ 2005.

Press F8 to jump to the first function; this is a very short function which calls two functions then returns, so our main function must be somewhere inside one of those function calls:

```
PUSH EBP
```

```
MOV EBP,ESP
```

```
CALL test_vc.00411050
```

```
CALL test_vc.00411860
```

```
POP EBP
```

```
RETN
```

Press F8 until you reach the first function call, then press F7 to step into it. Notice that we are back at the jump table, and that the actual function is located at `00412AB0`:

Press F8 again to follow the jump to the actual function. Here we see a lot of calls to functions like `GetCurrentProcessID` and `GetCurrentThreadId`, but no calls to any exit functions. The call to the main function is probably not located here, so press F8 until you return from this function, and you will be sitting at the second call which points to `00411860`. As before, press F7 to step into it, and notice that it does not take us to the jump table, but rather directly to the function itself.

Scrolling down in the function code, we see a call to `exit()`, and a few instructions above that a call to an internal function located at 00411136:

Set a breakpoint on the function call to 00411136 and press F9 to run the program. When it breaks, press F7 to step into the call; this again takes us to the jump table, so press F8 to get to the start of the function:

Now that looks more like it! There is a reference to our string (or at least part of it), and calls to `printf()` and `getchar()`. But getting here was kind of a pain, and there's an easier way. Notice how the call to the main function was made just before calls to `exit()` and `_cexit()`, and in the previous program that was compiled with GCC, the call to the main function was located just before a call to `_cexit()`? And remember how we were able to view all of the inter-modular calls? Well, right click the code window and again select 'Search for -> All intermodular calls':

Hmmm, there are two calls to `_cexit()` and one to `exit()`. Double click on the first call to `_cexit()` and Olly will take you to back to the code window and you will be located at 004118F6, which obviously is a call to `_cexit()`. Scroll up just a few instructions and there is the call to our main function located at 00411A01.

## EXE Patching

One of the great functions of a decent debugger is that it allows you to modify instructions and data, and Olly is no exception. `patchme.exe` is a simple program that prints the text "This is the wrong text!" to the screen, and is very similar to the `test.exe` program we worked with earlier. The only differences are that no source code is supplied for `patchme.exe`, and the object of this program is to make it print "This is the right text!". Depending on how the program was coded, there may be several different ways to accomplish this, so let's open it up with Olly and see what we can find.

Once opened in Olly, find the main function for `patchme.exe`; it should look like this:

---

As we've seen previously, everything through the two function calls at 004012B0 and 004012B5 is generated by the

compiler; the real code starts at 004012BA, where zero is moved into a local variable (we'll call it var1). Next, the memory address 0040301A is moved into another local variable, var2. In the memory dump window, go to the address 0040301A and you will see that it is our "This is the wrong text!" string; so we know that var1 is set equal to zero, and var2 is a pointer to the text string. Next, var1 is compared to 1, and if they are not equal (which we know they aren't), the program jumps down to 004012E5, where var2 is pushed onto the stack and printf() is called.

But look, there are two calls to printf(); because var1 didn't equal 1, the first one was skipped. Looking at the next instruction after the conditional jump (JNZ), we see that the memory address DS:[00402000] is placed into EAX, and then EAX is placed on the stack as an argument to printf(). To find out what is located at DS:[00402000], set a breakpoint on 004012BA (where var1 is set equal to zero), restart the application and press F9 to run it. When Olly breaks at 004012BA, press the space bar to change the assembly code:

Change the 0 to a 1, click 'Assemble' and then 'Cancel' to close the window. Now that we have changed the code to set var1 equal to 1, F8 through the next few instructions and notice that this time, the jump to 004012E5 is not taken. When you get to the 'MOV EAX,DWORD PTR DS:[402000]' instruction, notice that the information bar below the code window indicates that DS:[00402000] is the address 00403000. Take a look at this address in the memory dump window:

Well how about that...the string we want to print ("This is the right text!") is located there, just above the "wrong text" string. So, if var1 is set equal to 1, then the "right text" string is printed...to test this, press F9:

Bingo! Now if we want to permanently patch the program, all we have to do is go back into Olly, right-click the code window, select 'Copy to executable -> All modifications' and click the 'Copy All' button when prompted. Right click inside of the new window that appears and select 'Save File'. Save it as patched.exe, exit Olly, and run the new program - you'll be greeted with the "right text" string.

Intro to Keygens and Un-packing  
Writing Key Generators

Why write a keygen? Because while patching a file is useful, you don't learn nearly as much about reverse engineering than you do when writing a key generator. In many cases it is not desirable to permanently patch an executable, so it is necessary to completely reverse a function or algorithm; writing key generators is good practice in performing in-depth code analysis.

For this part of the tutorial, we will be examining KeygenMe3 by Tanatos, and ARealApp.exe (both included in the apps directory). In KeygenMe3, we will learn how to un-pack a compressed executable, do some serial fishing, and trick the program into becoming its own key generator.

ARealApp is a "keygen-me" program which uses a key generation algorithm that I reversed from a real-world shareware application. For legal purposes I did not want to identify the shareware program, so I wrote ARealApp using the exact same algorithm used in the shareware executable. Here we will completely reverse engineer the algorithm used for key validation, and write our own keygen in C.

### Executable Compression & Dumping

Let's focus first on KeygenMe3 first; open it up in Olly, click through any warning messages, and take a look at what we have:

As we saw in part one of this tutorial, most functions begin with a prelude which manipulates the EBP and ESP values accordingly; this program starts out with the PUSHAD instruction which pushes the contents of all the registers onto the stack. This is a sure sign of a packer, usually UPX or ASPack (opening the executable with PEiD does in fact identify this program as being packed with UPX). If you are unfamiliar with packers, they simply compress the executable file so that it takes up less space on the disk, and add a de-compression routine to the beginning of the executable in order to un-compress the program when it is loaded into memory. The basic layout of most of these simple routines is:

PUSHAD

*/\*de-compression routine here\*/*

POPAD

JMP OEP

We want to find the jump to the OEP (Original EP) so that we can view the original assembly code; since this jump is made only after the executable code has been decompressed, we will have no trouble viewing it. We could trace through the de-compression routine, but there's an easier way. Notice that the first instruction is PUSHAD (which saves all the register values onto the stack), and the last instruction before jumping to the OEP is POPAD (which restores the previously saved data from the stack to the registers). The routine does this to ensure that the register values will be set

up properly for the program; in other words, it doesn't want the original code to have any idea that extra code has been run before arriving at the OEP, because the instructions there are expecting certain values, and wouldn't know how to handle any errors that might result from variations in the register values. This means that the only instruction that will touch the values placed on the stack by PUSHAD is the final POPAD instruction. If we place a hardware breakpoint on those values after they have been pushed, then the next time they are accessed (which will be by the POPAD instruction), we will be sitting at the JMP OEP instruction.

Press F8 to execute the PUSHAD instruction; notice how the value of ESP changes to reflect the fact that new data has been pushed onto the stack:

Right-click the ESP value, and select "Follow in Dump". The memory dump window now shows the values at that address:

The first four bytes are the same bytes located in the EDI register, since it was the last register to be pushed onto the stack by the PUSHAD instruction. Note also that each byte of each register has been pushed in reverse order. Now, select those first four bytes in the dump window, right click and select "Breakpoint -> Hardware, on access -> Word". Press F9 and Olly will break right after those four bytes are accessed by the POPAD instruction:

Press F8 to execute the JMP instruction (this is the jump to the OEP that we were looking for!), and the code should look much more friendly now:

Now that we have let the program un-compress itself, we will save it in its un-compressed state. The easiest way to do this is with the OllyDump plugin. Right click the code window and select "Dump debugged process". In the window that pops up, click 'Dump', and save the file as tknm3\_dump.exe:

Olly will correct the entry point address, import function addresses, and anything else that needs to be fixed; run `tknm3_dump.exe`, and it should run normally.

If you are not planning on permanently modifying a program, it is not necessary to actually save it in its de-compressed form. However, since we will need to make modifications to this executable, this final step was required.

### Self-Keygens

Self key generation is the act of patching the target application in such a way that it reveals the correct serial key to the end user. This is most often done by finding the memory address that the calculated correct key is stored at, and replacing the text in the standard "Incorrect serial number" message box with the correct serial number. And that is exactly what we will be doing with `tknm3_dump.exe`.

But I'm getting ahead of myself. Let's assume that we know nothing of this program (other than the original executable was packed with UPX of course); the first thing we should do is open it up in Olly and run it! We have a dialog box asking for our name and registration key:

Clicking the 'Check' button without entering any information results in a message box telling us to enter a name that is at least three characters long; this is interesting, because it means that the serial is probably generated based on the name entered. So, put in a three character name (I used 'aaa') and click the 'Check' button again. What's this? We get the same error message. In fact, we have to enter a name that is at least four characters long in order to pass this first check - obviously a programming error, and it is always good to note such errors as they may come in useful while disassembling the code (in this case, it doesn't however).

Now that we've entered a valid name, we get another message box telling use we forgot to enter a serial number. So, put in something random like '12345' and click 'Check' again. We now get a "Wrong serial! Try again" message. Now, this message box is obviously the negative result of running our entered serial number through the serial validation algorithm, so if we can find where this message box was called from we can trace back from there and find the algorithm and/or comparison in question. In Olly, press F12 to pause the program, then press Alt+K to view the call stack. Here we see recently called functions, among them, two calls to `MessageBox`:

The second one, called from `00401172`, is the one we are looking for. We know this because anything with very high memory addresses, like `77D8054B`, is usually located in a DLL file, and the second message box was called from a memory address that resides inside the target program's memory range. So back in the code window of Olly, press `Ctrl+G` and go to address `00401172`:

Interesting, no? A few instructions above the address in question, we see there is a call to `GetDlgItemTextA`, followed by a call to `0040120D`. The return value of `0040120D` is compared using a `TEST` operation, and if `EAX` is zero, the serial error information is pushed to the message box function; if it is not zero, then the "Registered" text is pushed to the message box function. The function at `0040120D` definitely warrants some more investigation, but let's look at a few more things first. If you scroll up a little further, you'll see that our entered serial number is stored at `0040B420`, our entered name is located in `0040B1E0`, and there is an unknown number (it will be '3825205248' if you have used the name 'aaaa') stored in `0040B300`. Now, let's set a breakpoint (F2) at the first argument that is supplied to the `GetDlgItemTextA` call ('PUSH 32' located at `0040112F`), and restart the program in Olly.

Run the program again, supplying the same name and registration key, and click 'Check'. The first item that is pushed is the number `0x32` - this is the maximum size of the string, so we know that our serial can't be more than 50 characters long. The next item that is pushed is the address to store the string in, and you can see that this value is indeed `0040B420`, where we found our serial number stored the last time. The other two values are the control ID and dialog box handle, neither of which are important for our purposes. Step through the code until the call to `GetDlgItemTextA` has been executed. The next three instructions are important: there are two values pushed onto the stack, and then a call to the mysterious function located at `0040120D`. Looking at the two values that are pushed, the first is `0040B300` (remember that "strange" string we saw earlier?) and `0040B420`, which is the address where our entered registration key is located. Press F7 to trace into the function call.

This is a small function, consisting of a single loop:

First, the unknown string address is moved into `EDX`, while the address of our serial number is moved into `ECX`. The value of `ESI` is pushed onto the stack, and `ESI` is then set to 1. `EDX+1` is then loaded into the `EAX` register; this is interesting because it means that the first character of the unknown string will NOT be examined. Then, the SECOND byte of our registration key is moved into `DL` (again, the first character is ignored), and `DL` and the first character of the unknown string are compared. If they do not match, then the function jumps down to `00401231` where `EAX` is set to zero, the original value of `ESI` is restored, and the function returns. However, if they do match, then `ESI` is incremented, `EAX` is incremented, and if `ESI` is not greater than 4, then the function loops back to `0040121E` where the next two bytes in each of the strings are compared with each other. If `ESI` is greater than 4, then `EAX` is set to 1, `ESI` is restored, and the function returns.

If that seemed a little confusing, don't worry, it's just my lack of communication skills; the function is quite simple actually. The relevant pseudo code looks like this:

```
//EDX points to the unknown string
```

```
EDX = *unknown_string
```

```
//ECX points to the serial number we entered
```

```
ECX = *entered_serial
```

```
//ESI is initialized to one
```

```
ESI = 1
```

```
//EAX points to EDX+1, aka, the unknown string starting at the SECOND byte
```

```
EAX = EDX+1
```

```
/*this next instruction is just some math to make it easier to reference the unknown_string and the entered_serial...the unknown string is at the memory address stored in EAX, and the entered_serial is at EAX+ECX. Also, since EAX actually points to the second byte of unknown_string, EAX+ECX actually points to the second byte in entered_serial as well. */
```

```
ECX = difference between ECX and EDX
```

```
//loop four times
```

```
while ESI <= 4
```

```
    //do the bytes in unknown_string and entered_serial match?
```

```
if (ECX+EAX) == EAX
```

```
    //increase ESI for loop purposes
```

```
    ESI++
```

```
    //increase EAX, aka, look at the next byte in each string
```

```
    EAX++
```

```
else
```

```
    //if they don't match, set return value to 0 and return
```

```
    EAX = 0
```

```
    return
```

```
end
```

```
end
```

```
//if we've gotten here, then all comparisons were successful, so set return value to 1 and return
```

```
EAX = 1
```

return

To recap, we know the following:

- If the return value of 0040120D is 0, then we get the bad serial message; else, we get a good serial message.
- 0040120D compares our entered serial with a string that is located at 0040B300.
- 0040120D ignores the first byte of both our serial number and the string at 0040B300.
- Only bytes 2-5 in each string are compared (the loop only runs four times).

So, we can conclude that the valid serial number is located at 0040B300, but only bytes 2-5 are relevant. So, a valid serial has to be at least 5 bytes long; the first byte can be anything we want, but the next four must match with the respective bytes in the string at 0040B300. Note the string value at 0040B300 (I used a name of 'aaaa', so my value is '3825205248'). Restart the program, enter the same name, and for the registration key enter the first five numbers of previously noted value, replacing the first number with whatever you like (in my case I entered '08252'):

Bingo! Now we just need to patch a few instructions in order to turn this program into a handy little key generator for itself.

We know that the registration key is located at 0040B300, so all we have to do is change the memory address pushed to the MessageBox function from 0040A0A4 ("Wrong serial! Try again") to 0040B300 (the correct serial number):

This way, when an incorrect serial number is entered, you will get a message box that shows you the correct serial number:

Save your changes to the executable and the keygen is done!

## Reversing Key Generation Algorithms - Writing a Real Key Generator

Some people consider the previous keygen project cheating because we didn't actually reverse the key generation algorithm. If our only object was to create a keygen then this is fine, but from a purely RE perspective, they are right. So in this next installment, we will be completely reversing a registration key validation algorithm, and writing a key generator in C. The target application, USP.exe, has been written in C and uses a key-validation algorithm found in the wild (i.e., this was used in an actual shareware program). I have included the source code for both USP and the USP keygen, but don't peek at them until we've disassembled the binaries...that would take all the fun out of it.

First, let's run the USP to see what we have (yes, I was lazy and made it a CLI program). We see it prompts for a user name and a registration key...let's enter some fake credentials, say, 'test' for our name and '12345' for our key. We receive an "Invalid Registration Key!" message - not entirely unexpected. Now let's open USP up in Olly and see if we can find the code that accepts the user name and password, then subsequently validates (or in our case, invalidates) them. Since this is a simple program, we'll start by finding the main function, since there probably aren't too many nested functions we'll have to follow to find what we're looking for. In a real-world application we would probably have to use the trick we learned earlier of pausing the program then examining the call stack to find the section of code we are interested in. We find the main function at 00401380:

Note that the main function uses printf() and gets() to prompt for and retrieve the user name and registration key. Also of note is the function immediately above the main function (at 0040135D) which prints the "Invalid Registration Key!" text, then exits; this is our "bad" function. The gets() function takes one argument: the address to store the retrieved text into. Examining the above code shows that the user name we enter is stored at 004040D0, and the registration key at 004040B0. Strlen() is then called to determine the length of the registration key we entered (we know this, because the address where the registration key is stored is passed to the strlen() function), and the value returned is compared to 0x17 (23 in decimal notation). If the value in EAX is equal to 0x17, then a jump to 004013FC is made, but if not, then the function at 0040135D (the "bad" function!) is called. This tells us the first piece of information we need to know about the registration key; it must be exactly 23 characters in length.

The purpose of the rest of the code may not be so obvious when viewed in a dead listing, so let's see what happens when we enter a key of 23 characters in length. Set a breakpoint on 004013FC since we should land here if our key is the correct length, and run the program. Enter some fake credentials again, ensuring that your key is 23 characters long (I used '12345678901234567890123'), and Olly will pause at our breakpoint:

Now, this is interesting; the code is moving one byte of data from the memory address 004040B5 into the EAX register, then comparing that byte to the byte stored at 00402025. Now, we know that the registration key we entered is stored at 004040B0, so these instructions are actually comparing the 6th byte of our registration key with some other byte at 00402025. Taking a look at the information window just below the code window confirms that the byte pointed to is in fact the 6th byte of our registration key:

The question is, what is this byte being compared to? Press F8 to execute the MOVZX instruction, and again we can use the information window to see what is stored at 00402025:

Our 6th byte gets compared to a hyphen, and the following JNZ instruction shows that if they are not equal, then a jump is taken to 004014F3, which happens to be a call to the "bad" function at 0040135D. So we know that our 6th byte must be a hyphen - but look, this same comparison is performed on bytes 004040BB (the 12th byte in our registration string) and 004040C1 (the 18th byte in our registration string). This tells us that every 6th character in the string must be a hyphen. The registration key format now looks like:

12345-12345-12345-12345

Let's restart the program in Olly and enter in our new registration key ('12345-12345-12345-12345') so that we can pass this validation check as well, and we see that we do. Now we find our selves at 00401435, where two local variables, EBP-C and EBP-4 are set equal to zero before entering a loop which also contains a nested loop:

As you can see, I've added some comments in Olly (this can be done by pressing the semicolon key). We have three local variables here, all integers: var1 and var2 are used to count the number of iterations performed by the primary and nested loops respectively; var3 is incremented each time the nested loop is executed, and based upon the value of var3, the function at 00401290 is called with different arguments. Note that var3 is also incremented each time the primary loop is executed, but only after the if statements.

We can also see that the primary loop is executed 23 times, and the nested loop is executed 5 times and is doing something with the registration key we entered. This is particularly interesting, because we know that a valid registration key is 23 characters long, and that it consists of four sets of 5-character strings, each set being separated by a hyphen.

We can assume then, that these loops are either performing some type of calculations on the key as a whole, or on each of the four hyphen-separated strings that compose the key, or both.

To get a better idea of what is going on, let's start single-stepping through the code and see if we can determine what type of calculations it is performing. The first few instructions in the primary loop serve only to check if var1 is greater than 22 and to set var2 equal to zero. The nested loop is rather interesting however; here is another screenshot where I have added some more notes regarding the nested loop:

There doesn't seem to be any data located at 00404090 (yet), but we know that the registration key we entered is stored at 004040B0. We also know that var2 and var3 are initialized to zero, and incremented each time the nested loop executes, and that the nested loop loops five times. Here is some pseudo code to help clear things up:

```
var2 = 0;
```

```
5.times do
```

```
  //move one byte from our registration key into a variable located at 00404090
```

```
  00404090[var2] = 004040B0[var3];
```

```
  //increment var2 and var3 so that the above instruction will point to the next byte in each string
```

```
  var2++;
```

```
  var3++;
```

```
end
```

So, the first time the nested loop executes, it places the first five characters (i.e., the first of the hyphen-separated sets) of our registration key into a variable at 00404090. Each time this loop is called var2 is set to zero, so the values stored in 00404090 from the previous loop iteration will be overwritten each time the loop is called. Remember that we previously noted how var3 was also incremented once after the nested loop finishes executing? That is because the program wants to skip the hyphens in our registration string. For example, let's say that our registration key is '12345-67890-ABCDE-FGHIJ'. The first time the nested loop executes, it will copy '12345' into the variable at 00404090, the second time it is called, it will copy '67890' into 00404090, etc.

It appears that the program examines each 5-character set in our registration key individually, so let's try and find out what it does with them. The rest of the primary loop consists of if statements, comparing the value of var3 (which indicates how many characters of our registration key have been copied into 00404090) to several hard-coded values:

Again, let's create some pseudo code to help understand what is going on:

```
if var3 == 5
```

```
func_at_00401290(43,string_at_00404090);
```

```
else if var3 == 11
```

```
func_at_00401290(23,string_at_00404090);
```

```
else if var3 == 17
```

```
func_at_00401290(17,string_at_00404090);
```

```
else
```

```
func_at_00401290(53,string_at_00404090);
```

```
endif
```

```
var3++;
```

```
var1++;
```

For each if statement, the function located at 00401290 is called, and takes two parameters: an integer value, and the address of the variable that contains one of the hyphen-separated strings that make up our registration key. However, depending on which of those hyphen-separated strings is currently located at 00404090, a different integer value is passed to the function at 00401290. For the first string set, the value 43 is passed; for the second, 23, the third, 17, and for the fourth and final set, 53.

Once this primary loop is complete, the code immediately prints the "Thank you for registering" message; but, we know that we do not get that message when we enter an invalid registration key, so something is happening in the function at 00401290 that prevents the loop from completing. Put a breakpoint on the first call to 00401290 and press F9. When Olly breaks, take a quick look at the stack values that have been pushed; you will see the value 0x2B, and the memory address 00404090, which contains the first five characters of our registration key (from here out this will be simply called "the string"). Press F7 to step into the function call:

Again I have added some notes in Olly to help follow the assembly code, however, the first thing I want to look at is the ASCII string located at 00402000:

```
"2YOPB3AQC VUXMNRS97WE0IZD4KLF GHJ8165T"
```

If you notice, this string is 36 bytes long and contains all of the alpha-numeric characters. If you examine the code in Olly, you see that we have a primary loop with a nested loop, very similar to the previous loops we examined. In fact, the loops themselves function almost identically as the previous loops, so the loop structure should be familiar enough for you to figure out on your own. I have labeled six global variables which are used throughout the loop (gvar1-6); gvar2 and gvar3 are used to count loop iterations for the primary and nested loops respectively. If you skip the examination of the loops temporarily, you will see that gvar6 must be evenly divisible by the numerical value passed to the function. This is why we never completed the previous loop: if gvar6 is not evenly divisible, then it calls our "bad" function, which prints the "Invalid registration key" text and immediately exits the program. The trick now is to find out how gvar6 is calculated.

The primary loop's only function here is to loop five times, once for each character in the string that has been passed to the function; the inner loop loops 36 times (recall how the ASCII string of alpha-numeric characters is 36 bytes long?). So we already have a pretty good idea that each character in "the string" is compared to each character in the ASCII string: this is done using the CMP instruction at 004012E2. If the characters are not equal, then the program simply jumps down to 00401236, increments gvar3 (the inner loop counter), then runs the next iteration of the loop. However, if they are equal, the values in gvar1 (which is initialized to zero before the loops) and gvar3 (the number of times the nested loop has run, aka, the position of the character in the ASCII string that matches the character in "the string") are added together, and the sum saved in gvar4. Next, gvar1 is multiplied by 8, then added to itself (i.e.,  $gvar1 + gvar1 * 8$ ) and the result stored in gvar5. Finally, gvar5 is multiplied by 4 and added to gvar4; the result is saved in gvar6. Now we know how gvar6 is calculated, but there is one more detail; after gvar6 has been calculated, gvar1 is set equal to gvar6. This means that subsequent loops will be affected, because gvar6 is not re-set to zero before other characters in "the string" are calculated. In other words, if "the string" was '44444', the resulting gvar6 value for each character in "the string" will be different because one of the variables used to calculate gvar6 (gvar1 to be precise) changes each time there is a match between the character in "the string" and a character in the ASCII string.

All of this is much easier to comprehend if you single-step through the code in Olly and examine the above notes I have taken on each instruction. However, to further simplify, here's some pseudo-C code:

```
int key_check_function(int divisor, char the_string[])

{

//initialize gvar1 to zero

gvar1 = 0;

//the primary loop which iterates through each character in the_string

for(gvar2=0;gvar2<5;gvar2++){

//nested loop that compares the current character in the_string to every character in ascii_string

for(gvar3=0;gvar3<36;gvar3++){
```

```
if(the_string[gvar2] == ascii_string[gvar3]){
```

```
    //if the two bytes match, calculate gvar6 for that particular byte
```

```
    gvar4 = gvar1+gvar3;
```

```
    gvar5 = gvar1+gvar1*8;
```

```
    gvar6 = gvar4+gvar5*4;
```

```
    gvar1 = gvar6;
```

```
    }
```

```
    }
```

```
}
```

```
//the final gvar6 value is divided by the divisor integer passed to the function
```

```
mod = gvar6 % divisor;
```

```
//if gvar6 is not evenly divisible, then call the "bad" function; else, return 0
```

```
if(mod != 0) {  
  
    bad_function();  
  
}  
  
return 0;
```

That's a bit much, so if you're still struggling with understanding how it all fits together, don't worry (I didn't get it right away either ;)...just step through the code in Olly some more, observe what is happening with the various values, and refer back to my Olly notes and/or the above pseudo code until you do.

Get it? Good, because now we are going to write a keygen for this. Unfortunately the mathematical force is not strong with me, so I couldn't figure out a way to create a reverse algorithm for this function; the problem of course is that you have to choose a value that is divisible by the particular divisor (remember how the `key_check_function` is called with different values for the divisor each time), then work backwards from there. It just seemed like way to much work...so instead, I wrote a keygen that brute forces each of the four 5-character sets in the registration key (remember that they are examined individually too). To do this, I generated a random set of four alpha-numeric characters, then iterated through each character in the ASCII string, checking to see if it would create a valid key set if plugged in as the fifth character. If so, then that value was used; if not, then another four character string was calculated, and the process was done over again. The resulting code is listed in the `usp_key_gen.c` file in the source directory.

## Discovering and Exploiting Buffer Overflows

Now that we have sufficient practice in reversing closed source programs, it's time to examine buffer overflows. For this final section of the tutorial, we will be exploiting `weird.exe` which is a crack-me program included with Security Warrior. This program, as well as others used in the book, is available for download from O'Reilly Publishing (it is also included in the apps directory in this tutorial's zipped file). It is assumed that you have a basic knowledge of buffer overflows and stack operations.

### Examining the Program

As usual, before doing anything else, let's run `weird.exe` and see what we are working with. We are prompted for a serial number, so enter whatever you like (I used the usual '12345'), and hit enter. We are taken to a new line and nothing else happens. Hit enter again and the program quits. Well, we didn't get much information to work with there, so let's take a look at it in Olly. Looking around in the code a bit, we find the function that prompts us for our serial located at 00401108:

We see that there are several references to ASCII strings...the first three we saw when running the program, but the fourth string reference (located at 00401177) looks like what we want to see when we enter a serial. Notice that it is pushed as an argument to the function at 00404B38, which is also the function used to print the first three strings to the screen - essentially the equivalent of printf(). Further, we see that the function at 0040479C is called immediately after prompting for our serial, so we can assume that this function is similar to gets(). However, examining the code that is executed between the call to 0040479C and the "w00! congrats" string reference, we see that there are no jumps to 00401177. In fact, all of the jumps explicitly jump over that string reference and the subsequent call to 00404B38. This means that even if we did enter a valid serial, we would never actually validate. So it looks like we'll have to find a way to take over the EIP and redirect the program's execution to the line at 00401177.

### Some Quick BOF Information

For those already familiar with buffer overflows, skip to the next part of this section. As a quick refresher for everyone else, here are some basic rules about buffer overflows:

- The primary goal is to manipulate the value stored in the instruction pointer (EIP). Depending on the position and usage of variables in the program, there may be other ways to exploit a buffer overflow, but this is the holy grail.
- The EIP register can not be directly manipulated. For example, you can't do something like 'pop eip', or 'mov eip,eax'. However, when a function is called, the function must know how to return to the code that called it. This is done by automatically pushing a return address (the next instruction to be executed) onto the stack whenever a function is called. When the called function returns (the RETN instruction), it automatically pops the return address off the stack and into the EIP register.
- The above is the basis of your standard stack-based buffer overflow; when user-supplied data is copied into a location on the stack without any bounds checking, we may be able to supply a sufficiently long string of data that will overwrite the return value that was pushed onto the stack when the function was called. When the function returns, it will return to whatever memory address we overwrote the original return value with.
- To accomplish this, remember how data is pushed onto the stack. As data is pushed onto the stack, the stack grows downwards, from high memory addresses to low memory addresses. However, space is allocated for local variables on the stack as well, and when those variables are filled with data, the data grows UP. Suppose we have a function that defines three local char variables (one byte each):

```
char a,b,c;
```

For example purposes, the program will allocate three bytes on the stack for these variables, and the stack will look like this:

Return address (4 bytes)

Saved EBP (4 bytes)

char a           (1 byte)

char b           (1 byte)

char c           (1 byte)

Now, if we write one byte to char c, then the byte allocated for the c variable will be filled. But if we try to write two bytes to c, then the c variable will get the first byte, and the second byte will overflow into char b's memory address. If we write three bytes, the third byte will overflow into char a's memory address. If we write seven bytes into c, then we will fill the four bytes that contain the saved EBP value; if we write 11 bytes into c, then we will also fill the four bytes that contain the return address. So you can see that while the stack grows down, data written to the stack grows up.

### Locating a Buffer Overflow

Before we can try to exploit a buffer overflow, we must first find one. We can start by looking for obvious vulnerabilities, such as the use of `gets()`, `strcpy()` and the like; this can be done by right clicking the code window in Olly and selecting 'Search for -> All intermodular calls'. This reveals some known calls, but nothing like what we are looking for; judging from the fact that proprietary functions have been used in place of `printf()` and `gets()`, the programmer has probably done this throughout the application.

Now let's examine avenues of attack; in other words, what data do we control? The only data that we control in this program is the serial number, so let's take a look at what is done with that serial number after we hit the enter key. We have already identified that the function at 0040479C retrieves the text that we enter, so let's breakpoint on that and step into it when Olly breaks:

As usual, I've included some notes. This function takes one argument, which is a memory address (the argument is located at `EBP+8`). The address of the argument is loaded into `EBX`; then the program jumps down to 004047C6, where another function at 0040408A4 is called; it is supplied with the memory address 0040D19C. This is the actual function that retrieves our registration text, and it places it in 0040D19C. But, we needn't examine this function, because there is something interesting happening immediately in the current function code. Note how the loop starts copying the retrieved text one byte at a time onto the stack? While the function at 0040408A4 placed the string in a memory address, this loop is copying it into the address that was passed to the function as its one and only argument. Taking a look at the stack, we see that the address that our string is being copied into is 0012FF40, and the return address for this function (ideally the one we want to overwrite) is located at 0012FF38:

As you step through the loop however, you will notice that the string text is growing away from the return address at 0012FF38. Why? Recall that the stack grows down, but data grows up...the function was given an argument that contained the memory address to copy the string into, but that memory address (0012FF40) is located above the return address of the current function. However, not all is lost; the stack also contains return addresses for other functions, and these are located above 0012FF40. Let's trace through this current function until we return. We are now back in the first function that we examined. Let's trace through until the end of that function, and stop when we get to the RETN instruction at 0040118C. Take a look at the stack window:

Located at 0012FF90 is the return address for this function. We need the return address to read 00401177 in order for the program to jump back up to the instructions that print the "w00!" text. Some quick math tells us that 0012FF90 is 80 bytes above 0012FF40, where our registration text begins to be copied into. So, the 81st byte would overwrite the first byte of the saved return address, the 82nd would overwrite the second byte, and the 83rd would overwrite the third, etc. We can test this by supplying 84 bytes of text as a registration key:

Now let's look at the stack when we reach the RETN instruction for the first function:

0x41 is the hex value for the ASCII letter 'A' - now that we know we can control the EIP through this return instruction, it's time to code an exploit.

We don't want the function to return to 0x41414141, we want it to return to 0x00401177, so the last four bytes will need to be changed so that when they overwrite the saved return address with this value. For this, we will write a simple perl script:

```
#print 'A' 80 times to fill up the space between 0012FF40 and 0012FF90
```

```
print "A"x80;
```

```
#print the last three bytes of our desired return address in little-endian order
```

```
print "\x77\x11\x40";
```

What's going on with that last line you ask? Why are there only three bytes, and why are they in reverse order? The x86 architecture uses what is called little-endian byte order; in other words, data is written to the least-significant bytes first. So, as the loop writes each byte of our string into the stack, the data is written in this order (assume that the string 'ABCD' has been entered):

00000000	No bytes copied
00000041	First byte (A) copied
00004241	Second byte (B) copied
00434241	Third byte (C) copied
44434241	Fourth byte (D) copied

As you can see, 'A', the first byte in our string, is copied into the lowest byte in the buffer. The second byte, 'B' is copied into the second-lowest byte, and so on. So, in order for our return address to read '00401177', we need to enter a string that supplies 0x77 first, 0x11 second, and 0x40 third. Which brings us to our second question, why are we ignoring the highest-order byte, 0x00? Well, all text strings in a program must be null terminated (i.e., they must end with 0x00); user-supplied strings are no different, so when you supply your registration key, the 0x00 is automatically added on to the end of the string for us!

OK, now, the test. Run the perl program:

And copy/paste the output into the input of weird.exe. Press enter twice, and:

Whew!! That's a lot of information for a beginner. So let's let that sink in for a while, and we'd be more than happy to continue the conversation in the forums. Be sure to use that little cartoon bubble at the top of this article to get there.

Hope you're enjoying this continuing series on programming for the non-coding security professional. As always, send me your feedback on what we've already done as well as topics you'd like to see covered in future articles.