



Patch Developer Manual

revision 243
(c) Opatch by ACROS Security, 2017
<https://0patch.com>



Contents

1. Introduction.....	3
2. Opatch Agent for Developers	3
3. Security Notes	4
4. Patches and Patchlets.....	4
5. Patching Guidelines.....	5
6. Injecting a Patchlet.....	7
7. Suitable Places for Injecting a Patchlet	7
8. Patchlet Import Table.....	9
9. Anatomy Of a Patch File	10
10. Patch File Keywords.....	12
11. Building A Sample Patch.....	14
12. Final Notes.....	21



1. Introduction

Welcome to the *crowdpatching* community! We're extremely happy about your interest in writing patches with Opatch. Whether you want to patch vulnerabilities or functional bugs, or you need some way to correct code flow during your reverse engineering efforts, Opatch aims to be the tool for you. We hope you'll use it to solve your, and many other people's problems.

This document will show you how to set up your environment for writing patches, explain basic mechanics of Opatch, and provide many guidelines and hints to get you started.

It is assumed that you are already familiar with Opatch Agent in terms of its user interface and general functionalities. If not, it is highly recommended that you read the Opatch User Manual available at https://Opatch.com/user_manual.htm.

2. Opatch Agent for Developers

Opatch Agent for Developers is a slightly modified version of *Opatch Agent* (which is meant for production) and also includes a toolset needed for building your own patches. Much like *Opatch Agent*, *Opatch Agent for Developers* gets updates from the Opatch server when a new version becomes available. You cannot have both *Opatch Agent for Developers* and *Opatch Agent* installed on the same computer at the same time.

Specifically, *Opatch Agent for Developers* differs from *Opatch Agent* in the following ways:

1. *Opatch Agent for Developers* comes with *Opatch Builder*, our tool for compiling Opatch source files (.Opp files) into patch blobs that can get immediately applied to your local processes.
2. *Opatch Agent for Developers* doesn't validate signatures on patch blobs before applying them to newly-launched or running processes. This allows you to create patches locally on your computer and also test them there without having them signed. In contrast, *Opatch Agent* requires a patch to have our valid signature before applying it. Signatures are still being validated on all agents for patches that get delivered from the Opatch distribution server – we don't want you to get pwned by someone breaking into the server ;)
3. *Opatch Agent for Developers* registers a Opatch icon for .Opp files so that you can visually identify your patch source files, and adds two actions to the Explorer menu for .Opp files: "Build Patch" and "Build+Debug Patch". More on these later.
4. *Opatch Agent for Developers* automatically sets breakpoints on patchlet JMPs in WinDbg using the .o command instruction, provided that WinDbg has the correct magic word set.



3. Security Notes

Being interested in writing your own patches, you are likely sensitive to the state of security of any software you install on your computer, and any risks it brings with it. As such, here are some things you need to know about *Opatch Agent for Developers*.

1. **Opatch Agent for Developers does not verify digital signatures for patches when applying them to local processes.** (If it did, you wouldn't be able to apply your own patches as you don't have the signing key.) As a consequence, local malware with administrative privileges¹ could store a malicious patch in the Opatch database and achieve its own persistence by using Opatch Agent for injecting malicious code into a system process such as `winlogon.exe`.
2. **Beware of malicious .Opp files.** Opatch Builder by design supports launching an executable specified in a `.Opp` file with a debugger. Inspect every `.Opp` file from an untrusted source before executing "Build + Debug" on it.
3. **Beware of debugging potentially malicious processes with "Build+Debug".** To allow for automatic setting of breakpoints, Opatch Builder instructs WinDbg to accept external commands from the debuggee via `.ocommand`. This means that a malicious debuggee (potentially running as a low-privileged user) could instruct WinDbg (potentially running as admin) to execute an external application with arbitrary parameters, thereby elevating its privileges. If you want to debug a malicious process, either debug it manually (not via "Build+Debug") or manually change the `.ocommand` magic word.
4. **Current update procedure doesn't (yet) preserve your "unofficial" patches in the local Opatch database.** Updating *Opatch Agent for Developers* (when an update becomes available) will delete your own patches from the local Opatch database. (Yes, we're working on that.) Should this happen, you can simply re-build the patches from your `.Opp` files after the Agent has been updated.

4. Patches and Patchlets

Opatch currently supports patches that inject X86 or X64 machine code at a desired offset in a Windows binary, and optionally jumps over a selected number of bytes to effectively remove one or more original machine code instructions in said binary.

Each patch applies to **exactly one** binary, namely the binary that has the exact crypto hash specified in the patch. (Note that you don't see this hash in the `.Opp` patch source file as patch file only specifies the path to the binary while Opatch Builder calculates the hash for you.)

A patch comprises one or more *patchlets*; each patchlet defines the code to be injected at a specific offset from the binary's base address, an optional number of original code bytes to jump over (to

¹ Yes, we know, local admin malware means game over anyway, but still...



implement removal of original code), and an optional list of functions to import from selected binaries in order to be able to call them from the patchlet code.

Some patches - like those for typical buffer overflow vulnerabilities - only need a single patchlet, while some others – like those for typical use-after-free vulnerabilities – need more than one.

Each patch has a globally unique ID, and each of its patchlets has a patch-wide unique ID. As a rule of thumb, patchlets should be identified sequentially with IDs 1, 2, 3, etc. While a patch ID needs to be globally unique when deployed to the Opatch distribution server for distribution to agents around the World, you can use any unused patch ID during local patch development. We usually give patches under development IDs above 10000 to avoid conflict with existing patches that arrive from the distribution server. (This will obviously have to be revised as the number of official patches starts to grow.)

5. Patching Guidelines

Writing a patch is a delicate endeavor. You will be changing existing machine code that was almost certainly generated by a compiler; this is good in terms of recognizing compilers' coding patterns, and bad because compilers heavily optimize the code and make it more difficult to match it to the source code (should you happen to have it).

You will first have to understand the nature and context of the bug you're about to patch to the point of being able to say: "Okay, I now know exactly what the problem is," and then find a way to reliably and efficiently fix the bug. There will generally be more than one way to fix the bug, and you will want to find the one that has the least impact on the original code while fixing the problem in its entirety without allowing ways to bypass it and – importantly! - without breaking anything. In general: the less patch code the better, the fewer patchlets the better.

Always keep in mind that you're a guest in a likely huge and complex code base that you can't possibly understand as well as its original developers, and your only job is to put a plug in a tiny hole without causing any problems to original inhabitants or making their existing problems worse. This means, for example:

1. If you change a CPU register or a local variable, you have to make sure to restore it to its original value before letting the original code continue – unless you can prove that the original code will not use that value any more. (E.g., if you change `ecx` and the original code executing after your injected patch code also changes `ecx` before ever using it, it's okay not to restore it.)
2. If you make a call to some function from your patch code, you need to either completely understand its side effects (e.g., modifying registers, using blocking operations that might cause deadlocks, taking time that could cause timeouts in some other code waiting for the patched code to finish, etc.) and prove that they are inconsequential, or make sure to



neutralize these side effects (e.g., by storing relevant registers on stack before the call and restoring them after the call).

3. Take as little space as possible. Your patch code must be trivial to review by anyone understanding the bug and looking at your patch. It's not called "micropatching" for nothing. Also, do write many comments in your patch code: any instruction can be decorated with a comment using a semi-colon.
4. Be extremely cautious with multi-patchlet patches! While we've designed Opatch to allow a patch to consist of more than one patchlet, you must be aware that it is inevitably *possible* (although perhaps extremely unlikely) that when your patch is applied to an already running process, some thread might end up executing some, but not all of your patchlets. For instance, imagine only two patchlets: patchlet A storing some value in registry `edx` (which happens not to be used anywhere else), and patchlet B subsequently taking the said value from registry `edx` and using it, assuming that patchlet A has previously stored the value in it. It can well happen that at the moment when the patch is applied, some thread's execution is between the two patchlets, meaning that once its execution is resumed, it will only execute patchlet B without having previously executed patchlet A. Obviously this can quickly lead to serious problems. So while you may find it tempting to pass data from one patchlet to another, or to make patchlets inter-dependent in some way, *you really must not*. **Each patchlet must be able to execute independently, whether other patchlets have been / will be executed, or not.**

Sometimes understanding the problem or figuring out the way to fix it will take you a long time (especially if you're not experienced in reverse engineering), and sometimes this will frustrate you and make you want to give up. We're planning to release lots of material to help you with both stages of patching (you can already find some on our blog at <https://Opatch.blogspot.com>), and we're building a patching community you'll be able to turn to for help.



6. Injecting a Patchlet

When Opatch Agent injects a patchlet into the original code, it overwrites 5 bytes of the original code with a 5-byte `JMP` instruction that transfers code execution to the patchlet code. This is a process well known from function hooking, where one or more original machine code instructions from the very beginning of a function are copied («relocated») to another place in memory (called a «trampoline»), while their original location is overwritten with a `JMP` instruction to injected code (in our case: patchlet code). The patchlet code ends with a `JMP` to the trampoline in order to execute the relocated original instructions, and the trampoline is then completed with a `JMP` back to the first original code instruction after the relocated instructions.

Opatch takes traditional hooking to a higher level by:

- 1) supporting the injection almost anywhere² in the code, not just at the beginning of functions;
- 2) removing (jumping over) any number of original code instructions after the injection point, allowing you to effectively replace existing code with your patch code, or remove flawed code.

7. Suitable Places for Injecting a Patchlet

For several reasons, a patchlet cannot be injected just anywhere in the code:

- 1) Some original instructions cannot be safely relocated to another address. For example, a short `JZ` with a single-byte operand at address `10000000h` cannot be relocated to address `20000000h` because the recalculated relative jump offset from the new address would not fit into that single byte. The same goes for two-byte jumps, so we can only relocate 4-byte (32-bit) jumps. This also applies to relative `CALL` instructions with 2-byte (16-bit) operands. (We're planning to provide support for all these cases in the future by replacing short relative jumps and calls with 32-bit alternatives that can reside anywhere in the memory.)
- 2) A call instruction can only be safely relocated if it is *the last* relocated instruction; it is possible that at the moment of patching, one of the threads would be inside a relocated call (possibly already several further calls deeper down the call stack) and when it returns, it has to return to the original instruction that was there when the call was executed. If the relocated call is the last relocated instruction, we can be sure that instructions immediately after it on its original location are intact and can be safely returned to. However, if another instruction after the call was relocated to trampoline, it would mean that this instruction was also at least partly overwritten by `JMP Patchlet`, so returning to it from a thread

² Not all original instructions can be relocated to a trampoline; refer to section «Suitable Places for Injecting a Patchlet».

PATCH

currently executing the relocated call would result in executing unwanted code, almost certainly causing functional problems. On the left image below is a suitable code block to inject at, on the right an unsuitable code block, because the call is not the last to-be-relocated instruction.

```
mov eax, [esi+08h]
call [eax]
```

```
call [eax]
cmp ecx, 00000001h
```

- 3) Original instructions that are a destination of any jump or call elsewhere in the original code can only be safely overwritten with our 5-byte `JMP Patchlet` instruction if they are *the first* relocated instruction (as that would result in the said jump or call transferring execution to `JMP Patchlet` which would be okay). Any non-first overwritten original instructions must not be a destination of any jump or call, as such jump or call would end up executing unexpected code in the middle of the `JMP Patchlet` instruction. We recommend using some powerful disassembler (e.g., IDA) to determine whether original instructions at your potential patch location happen to be a destination of any jump or call.

```
→ mov ecx, edi
   cmp ecx, esi
```

```
→ mov ecx, edi
   cmp ecx, esi
```

- 4) Absolute calls and jumps can be safely relocated.
- 5) 32-bit relative calls can be safely relocated as we're recalculating their offset operands to work at their relocated address.



8. Patchlet Import Table

If needed, a patchlet can make calls to functions - for instance, Windows API functions or functions in the module it is patching. This is done by having desired functions imported to the *Patchlet Import Table* (PIT) using the `PIT` keyword. Let's look at some examples of imported functions in a patchlet injected into `myapp.exe`.

```
PIT user32.dll!MessageBoxW,myapp.exe!0x7798
```

The above instruction makes two functions available to the patchlet (note their names start with "PIT_" to avoid confusion with local labels):

1. `PIT_MessageBoxW`: The `MessageBoxW` function exported from Windows' system library `user32.dll`, and
2. `PIT_0x7798`: Location at offset `0x7798` from `myapp.exe`'s base. This can be very useful in case your patchlet implements some sanity check (e.g., for excessive height or width of an image) and the appropriate response would be to leave the patched function: using an import like this allows your patch to simply jump to the function epilog instead of having to replicate said epilog, usually comprising several `POP` instruction, `ESP` manipulation and a `RET`. Note that while you can use this notation to reference non-exported functions/locations from any binary, it is only safe to reference locations from the binary you're patching, as this guarantees that you're using the correct version of binary. You wouldn't want to reference a non-exported function from some DLL that might be different on another user's system.

In addition, there is one other function that is always available for calling from patchlet code:

3. `PIT_ExploitBlocked`: If you call this function (it takes no arguments), `Opatch Agent` will display an "Exploit Attempt Blocked" popup to the logged-in user. While the dialog requires manual closing, this function is not blocking and returns as soon as it sends out an instruction to show the popup. (The popup is displayed by `Opatch Tray`.)

Some guidelines for using imported functions:

- Avoid calling functions if possible. Any function call is likely to significantly increase the amount of code executed by your patch.
- The most safe-to-use functions are those from the module that is being patched, either exported functions or functions you specify by offset from the module base. That module's code is namely always guaranteed to be the same on all computers, as the patch will not get applied if the module's hash does not match the hash specified in the patch.
- It may sound safe to import functions from the main executable of the patched process (e.g., you're patching `lib.dll` that gets loaded by `app.exe`, and you want to import a function from `app.exe`). But there are risks here: What if another executable also loads `lib.dll`



and your patchlet can't find an exported function from `app.exe`? Worse yet, what if you're importing a function from offset `0x888` from `app.exe`, and then an update replaces `app.exe` but leaves `lib.dll` intact? Your patchlet will still get applied to `lib.dll` and will call a "function" at offset `0x888` into `app.dll` – this will likely not be the function you intended to call but rather some random code.

- Even for the safest of function calls, you have to be sure that the binary you're importing a function from is already loaded when the module you're patching is loaded (so that the imported function will already be there), and that it will remain loaded until the process exits (so that you don't end up calling a function in an already-unloaded module, and crash).
- Be aware that Windows API functions may behave differently on different Windows versions (and even between service packs or monthly updates). Some functions may even only exist on newer Windows versions: `RemoveDllDirectory`, for instance, only exists on Windows 8 or later and Windows Server 2012 or later. Note that a patch will not get applied unless all imported functions from all its patchlets can be found.
- When a patched module is loaded, all its patches are copied to an internal cache, and at that time, all PIT addresses are calculated based on the current base of the modules they refer to. For example, if PIT includes `lib.dll!function`, the address of function in `lib.dll` is determined using `GetProcAddress` and stored to the PIT. This introduces some risks you need to consider:
 - If `lib.dll` gets unloaded at a later time, PIT will point to an invalid address and executing the patch code will result in a crash.
 - If `lib.dll` is not yet loaded in the process when we find it in PIT, `Opatch` loader will force load it – which will result in its `dllmain()` function getting executed, provided it has one. This may have unexpected results as this function may assume it will only get executed after some other initialization has taken place, and that may not be the case now.
 - If `lib.dll` is not yet loaded in the process when we find it in PIT, `Opatch` loader calling `LoadLibrary("lib.dll")` may result in a binary planting vulnerability.

9. Anatomy Of a Patch File

A `.Opp` patch file is formatted in the following way:

- **Patch data** section: This section specifies the main parameters for the patch, such as patch ID, the binary it patches, the vulnerability ID, and whether it's a 32-bit or 64-bit patch. This section is followed by one or more patchlet sections.
- **Patchlet** section: This section, beginning with `patchlet_start` and ending with `patchlet_end`, specifies two things:
 1. the main parameters for the patchlet, such as patchlet ID, offset for injecting the patchlet code, optional number of bytes of original code to jump over and optional imported functions the patchlet code is going to call;



- patchlet code, beginning with `code_start` and ending with `code_end`, contains patchlet's assembly code in the NASM³ format

The following image shows an actual `.Opp` file for the *Foxit Reader FlateDecode Use-After-Free* vulnerability ZDI-16-392⁴. You can see that it begins with a Patch data section, which is followed by two Patchlet sections. (We patched this use-after-free vulnerability by sabotaging the *free* and marking the not-freed buffer with a "BADBAFFA" marker, then catching this marker at *use* and preventing its use.)

```
MODULE_PATH "C:\Program Files (x86)\Foxit Software\Foxit Reader\FoxitReader.exe"
PATCH_ID 250
PATCH_FORMAT_VER 2
VULN_ID 1448
PLATFORM win32
```

Patch data

```
patchlet_start
PATCHLET_ID 1
PATCHLET_TYPE 2
PATCHLET_OFFSET 0x00493AB6
N_ORIGINALBYTES 5
JUMPOVERBYTES 2
```

Patchlet 1

```
code_start

jz RETURN ; if ecx is 0, return
; (remember that the last instruction before entering this patch
; code was "test ecx, ecx", and that the JZ we removed was just
; going to jump to retn, which we can do here as well
cmp dword [ecx+8], 0xBADBAFFA ; does ecx point to a previously "freed" buffer?
jz RETURN ;if so, return
jmp RESUME
RETURN:
retn
RESUME:

code_end
```

Patchlet code

```
patchlet_end
```

```
patchlet_start

PATCHLET_ID 2
PATCHLET_TYPE 2
PATCHLET_OFFSET 0x00495663
N_ORIGINALBYTES 5
JUMPOVERBYTES 5; we eliminate the 5-byte call to free as we will write BADBAFFA to it

code_start
mov dword [esi], 0xBADBAFFA ; instead of freeing esi, we write BADBAFFA to it
code_end

patchlet_end
```

Patchlet 2

³ <http://www.nasm.us>

⁴ <http://www.zerodayinitiative.com/advisories/ZDI-16-392/>



10. Patch File Keywords

Patch Data		
Keyword	Mandatory	Description
RUN_CMD	No	Full path to the executable you want to have launched when selecting “Build+Debug” from the shortcut menu on a .Opp file. Opatch Builder will launch this executable in WinDbg debugger.
MODULE_PATH	Yes	Full path to the binary to be patched. Opatch Builder calculates a crypto hash from the content of this file and stores both filename (without path) and this hash to the patch blob. The patch will only be applied to binaries with the same name and the same hash.
PATCH_ID	Yes	Unique identifier for this patch. If a patch with this ID already exists in your local Opatch Agent’s database, that patch will be overwritten with this one when you build it.
PATCH_FORMAT_VER	Yes	The only supported format version at this time is 2 .
VULN_ID	Yes	ID of the vulnerability this patch is fixing. If a vulnerability with this ID exists in your local Opatch Agent’s database, its title and CVE ID will be shown in the Opatch Console and on Opatch popups. If you don’t know this ID, we recommend using some arbitrary large value such as 10000.
PLATFORM	Yes	win32 for 32-bit binaries or win64 for 64-bit binaries

Patchlet Data		
Keyword	Mandatory	Description
PATCHLET_ID	Yes	Unique identifier for this patchlet inside the patch. Make sure that each patchlet in a patch has a different ID; we recommend using 1, 2, 3...
PATCHLET_TYPE	Yes	The only supported patchlet type at this time is 2 .
PATCHLET_OFFSET	Yes	Offset from the base of the module where the patchlet is to be injected. The patchlet gets injected before the instruction



		at this offset, while the said instruction (and if needed, subsequent instructions) gets relocated to another place in memory where it will be executed after the patchlet code. This value can be either in hex (0xAAAAAAAA format) or decimal (AAAAAAAA format). We recommend using hex format.
N_ORIGINALBYTES	No	The number of original bytes at the PATCHLET_OFFSET location that get verified before the patchlet is applied. Default value is 5, which is all bytes overwritten by our “jump to patchlet” instruction. The only use cases for setting this value we know of are: <ol style="list-style-type: none"> 1) injecting at a location of a JUMP or CALL that gets relocated by Windows according to the PE relocation table (in this case we set N_ORIGINALBYTES to 1 as only the instruction code is constant); 2) injecting at a location where the to-be-patched code does not exist in its final form at module load time – e.g., is either decrypted or decoded after the module loads (in this case we set N_ORIGINALBYTES to 0).
JUMPOVERBYTES	No	The number of bytes of the original code we want to jump over (i.e., effectively remove) after the execution of patchlet code is completed. Default value is 0, which means we want to keep all of the original code. If you set this value, you must make sure that the number of bytes you specify corresponds to the actual length of original instructions from the PATCHLET_OFFSET location forward. For instance, you can use JUMPOVERBYTES 3 for jumping over two instructions <code>xor eax, eax</code> (2-byte instruction) and <code>inc esi</code> (1-byte instruction) located at PATCHLET_OFFSET.
PIT	No	Patchlet import table – allows you to specify exported functions or offset-based locations, either in the module you’re patching or some other module. Format is: <pre>PIT <module_name1>!<function_name_or_offset1>, <module_name2>!<function_name_or_offset2>, ...</pre> See section 9 for more information.
code_start code_end	Yes	Non-empty patchlet code (to be injected right after the original code instruction at PATCHLET_OFFSET) must be located between these two keywords; the code must be in NASM format as it’s being compiled by NASM.



11. Building A Sample Patch

This section will guide you through the process of building a sample patch. You'll need the following setup before you begin:

- 1) **A Windows computer with one of the following Windows versions⁵:**
 - a. Windows 10 64-bit
 - b. Windows 8.1 64-bit
 - c. Windows 7 64-bit
 - d. Windows 7 32-bit
 - e. Windows XP 32-bit

- 2) **The latest version of *Opatch Agent for Developers*** must be installed and registered on your computer. You can download the agent installer from <https://dist.Opatch.com/download/latestagentdev>. After successful installation, you will be prompted to register your agent when the Opatch Console is launched for the first time. Use your existing Opatch account credentials if you already have one, or register a new account at <https://dist.Opatch.com/User/Register>.
(Note that if you currently have *Opatch Agent* installed, you will need to manually uninstall it and install *Opatch Agent for Developers*; while they share much of the code base, these are two distinct products. If unsure about which Agent you have installed, look at its version number: if the last five digits look like "2xxxx", it is *Opatch Agent for Developers*, otherwise it is the production *Opatch Agent*.)

- 3) ***Opatch Agent for Developers sample package*** must be unpacked on your computer in a folder of your choice. The package can be downloaded from <https://Opatch.com/files/DevAgentSamplePackage.zip>.

- 4) **WinDbg must be installed** if you want to use the "Build+Debug" feature. On 64-bit systems, we recommend installing both 32-bit and 64-bit WinDbg. We are officially supporting the following WinDbg versions⁶:
 - a. WinDbg 6.12.2.633 on pre-Windows 7 systems
 - b. WinDbg 6.3.9600.16384 on Windows 7 and newer systems

- 5) **System-wide WinDbgDir_ environment variable(s)** must be set if you want to use the "Build+Debug" feature:
 - a. On 32-bit and 64-bit systems, WinDbgDir_x86 environment variable must be set to the directory of 32-bit windbg.exe; for the default installation location of 32-bit WinDbg 6.3.9600.16384, you can use the following command to set this variable:

⁵ Other Windows versions back to Windows XP and Windows Server 2003 likely work as well, although we have experienced problems with deploying a patch to registry on Vista.

⁶ Other WinDbg versions are likely to work well too, but there may be some differences in options, flags or features that could cause incompatibility issues.



```
setx WinDbgDir_x86 "C:\Program Files (x86)\Windows  
Kits\8.1\Debuggers\x86"
```

- b. On 64-bit systems, WinDbgDir_x64 environment variable must be set to the directory of 64-bit windbg.exe; for the default installation location of 64-bit WinDbg 6.3.9600.16384, you can use the following command to set this variable:

```
setx WinDbgDir_x64 "C:\Program Files\Windows  
Kits\8.1\Debuggers\x64"
```

Having all the above, you're ready to start. Let's go!

Step #1: Launch SmokeTest_x86.exe

SmokeTest_x86.exe (part of the sample package you have downloaded) is our sample executable that does one thing only: it pops up a "Hello World!" message box. Double-click this executable and notice the message, then close it.

Step #2: Build a patch

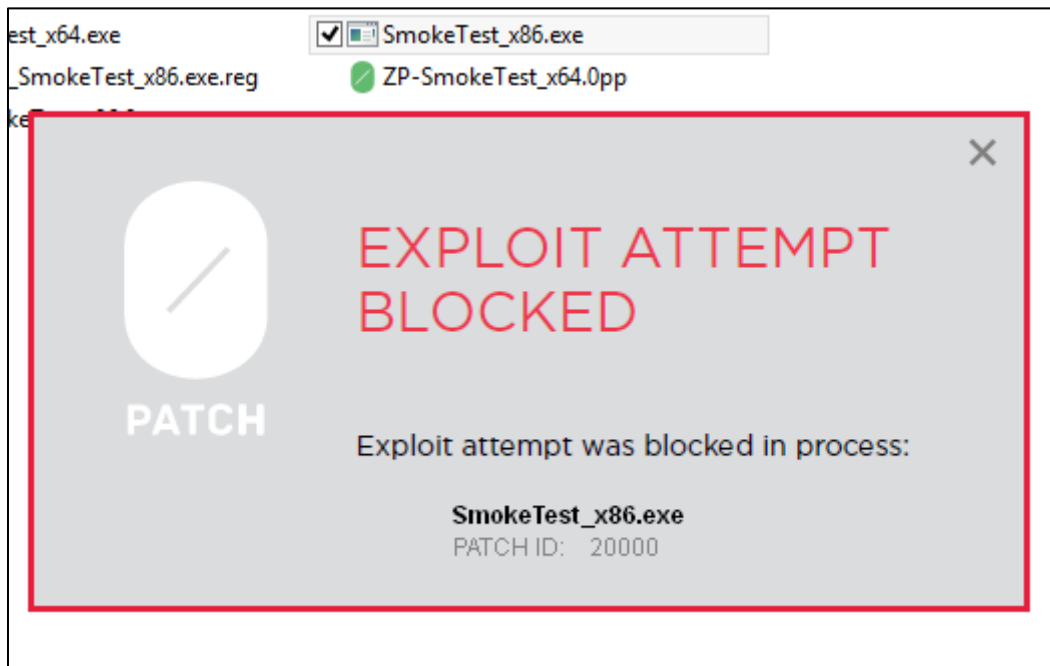
ZP-SmokeTest_x86.0pp is a patch source file for patching SmokeTest_x86.exe by simply injecting a call to PIT_ExploitBlocked, which should result in displaying a 0patch "Exploit Attempt Blocked" popup when the test executable is executed.

Right-click on ZP-SmokeTest_x86.0pp and select "Build Patch". A command interpreter (cmd.exe) window shortly appears, then a UAC prompt is displayed asking your permission to launch Registry Editor. Confirm the UAC prompt. The command interpreter window closes.

If everything went well, you have just created a patch with ID 20000 for SmokeTest_x86.exe. Open the 0patch Console and find patch #20000 at the end of the list in the "PATCHES" tab. You'll be able to disable and enable this patch there from now on.

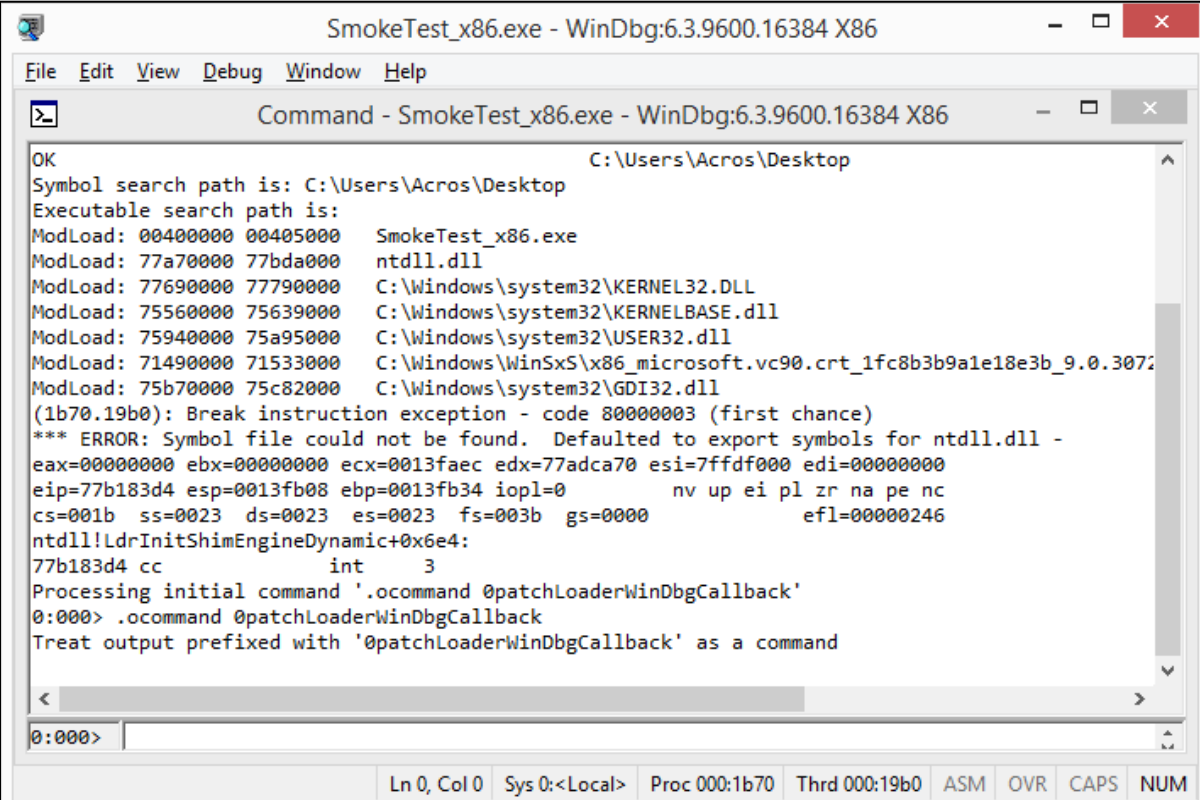
Step #3: Launch patched SmokeTest_x86.exe

Double-click `SmokeTest_x86.exe` and notice that, in addition to the “Hello World!” message box, an “Exploit Attempt Blocked” popup also appears as a result of your patch injecting a call to `PIT_ExploitBlocked` in the code of the testing executable.



Step #4: Build + Debug a patch

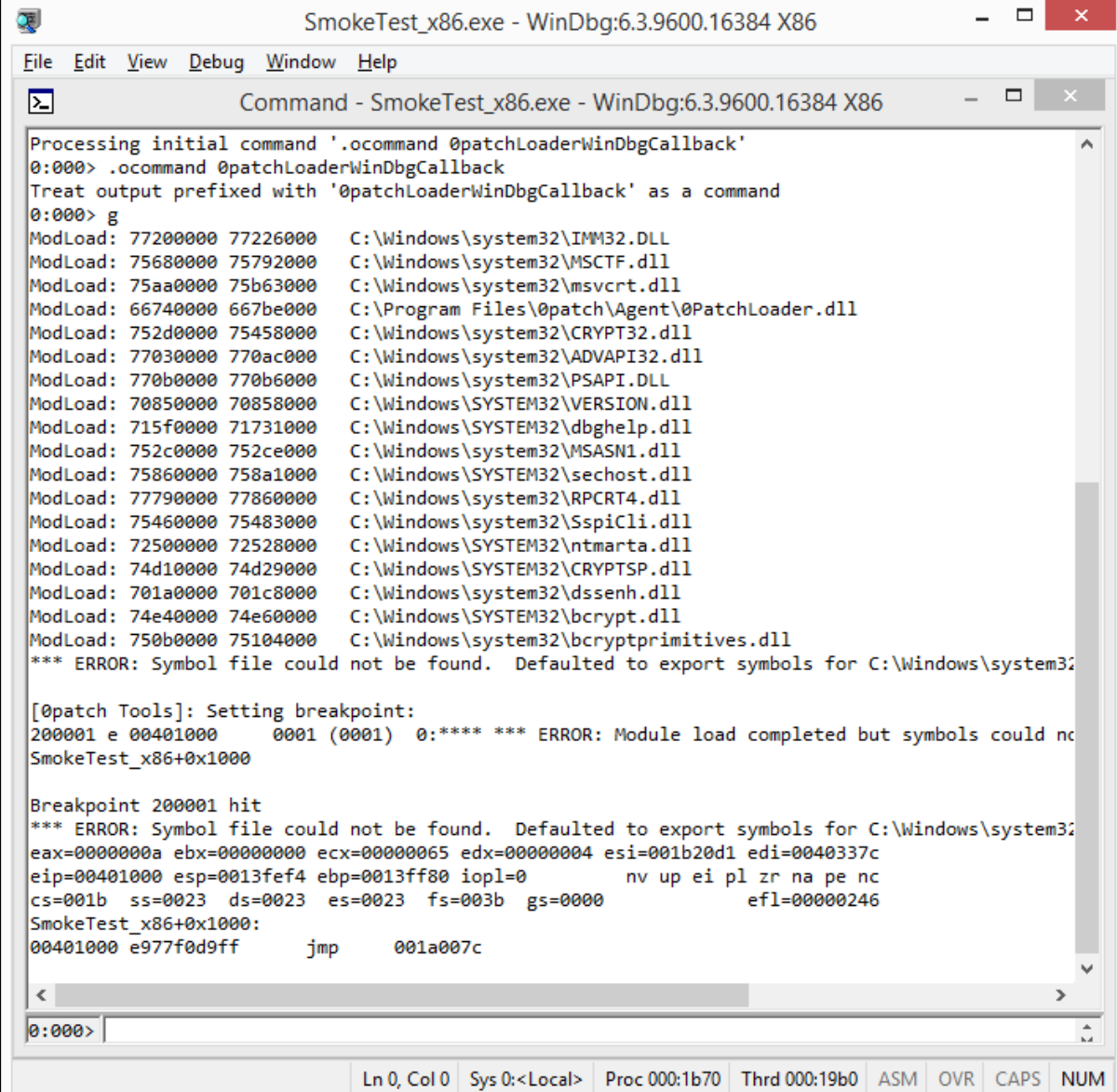
Right-click on `ZP-SmokeTest_x86.0pp` and select “Build+Debug Patch”. A command interpreter (`cmd.exe`) window shortly appears, then a UAC prompt is displayed asking your permission to launch Registry Editor. Confirm the UAC prompt. The command interpreter window closes and WinDbg is launched, attached to a newly-launched `SmokeTest_x86.exe`, as shown on the image below.



```
SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
File Edit View Debug Window Help
Command - SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
OK C:\Users\Acros\Desktop
Symbol search path is: C:\Users\Acros\Desktop
Executable search path is:
ModLoad: 00400000 00405000 SmokeTest_x86.exe
ModLoad: 77a70000 77bda000 ntdll.dll
ModLoad: 77690000 77790000 C:\Windows\system32\KERNEL32.DLL
ModLoad: 75560000 75639000 C:\Windows\system32\KERNELBASE.dll
ModLoad: 75940000 75a95000 C:\Windows\system32\USER32.dll
ModLoad: 71490000 71533000 C:\Windows\WinSxS\x86_microsoft.vc90.crt_1fc8b3b9a1e18e3b_9.0.3072
ModLoad: 75b70000 75c82000 C:\Windows\system32\GDI32.dll
(1b70.19b0): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ntdll.dll -
eax=00000000 ebx=00000000 ecx=0013faec edx=77adca70 esi=7ffdf000 edi=00000000
eip=77b183d4 esp=0013fb08 ebp=0013fb34 iopl=0          nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrInitShimEngineDynamic+0x6e4:
77b183d4 cc          int     3
Processing initial command '.ocommand @patchLoaderWinDbgCallback'
0:000> .ocommand @patchLoaderWinDbgCallback
Treat output prefixed with '@patchLoaderWinDbgCallback' as a command
0:000>
Ln 0, Col 0 Sys 0:<Local> Proc 000:1b70 Thrd 000:19b0 ASM OVR CAPS NUM
```

PATCH

Make sure the focus is on WinDbg's Command window and press F5; WinDbg should continue with execution and stop on breakpoint 200001 at a JMP instruction, as shown on the image below. This is the exact JMP instruction Opatch Agent has put in the original code to inject the patchlet code.



```
SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
File Edit View Debug Window Help
Command - SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
Processing initial command '.ocommand @patchLoaderWinDbgCallback'
0:000> .ocommand @patchLoaderWinDbgCallback
Treat output prefixed with '@patchLoaderWinDbgCallback' as a command
0:000> g
ModLoad: 77200000 77226000 C:\Windows\system32\IMM32.DLL
ModLoad: 75680000 75792000 C:\Windows\system32\MSCTF.dll
ModLoad: 75aa0000 75b63000 C:\Windows\system32\msvcrt.dll
ModLoad: 66740000 667be000 C:\Program Files\@patch\Agent\@PatchLoader.dll
ModLoad: 752d0000 75458000 C:\Windows\system32\CRYPT32.dll
ModLoad: 77030000 770ac000 C:\Windows\system32\ADVAPI32.dll
ModLoad: 770b0000 770b6000 C:\Windows\system32\PSAPI.DLL
ModLoad: 70850000 70858000 C:\Windows\SYSTEM32\VERSION.dll
ModLoad: 715f0000 71731000 C:\Windows\SYSTEM32\dbghelp.dll
ModLoad: 752c0000 752ce000 C:\Windows\system32\MSASN1.dll
ModLoad: 75860000 758a1000 C:\Windows\SYSTEM32\sechost.dll
ModLoad: 77790000 77860000 C:\Windows\system32\RPCRT4.dll
ModLoad: 75460000 75483000 C:\Windows\system32\SspiCli.dll
ModLoad: 72500000 72528000 C:\Windows\SYSTEM32\ntmarta.dll
ModLoad: 74d10000 74d29000 C:\Windows\SYSTEM32\CRYPTSP.dll
ModLoad: 701a0000 701c8000 C:\Windows\system32\dssenh.dll
ModLoad: 74e40000 74e60000 C:\Windows\SYSTEM32\bcrypt.dll
ModLoad: 750b0000 75104000 C:\Windows\system32\bcryptprimitives.dll
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\Windows\system32\
[Opatch Tools]: Setting breakpoint:
200001 e 00401000 0001 (0001) 0:**** *** ERROR: Module load completed but symbols could not be found for
SmokeTest_x86+0x1000

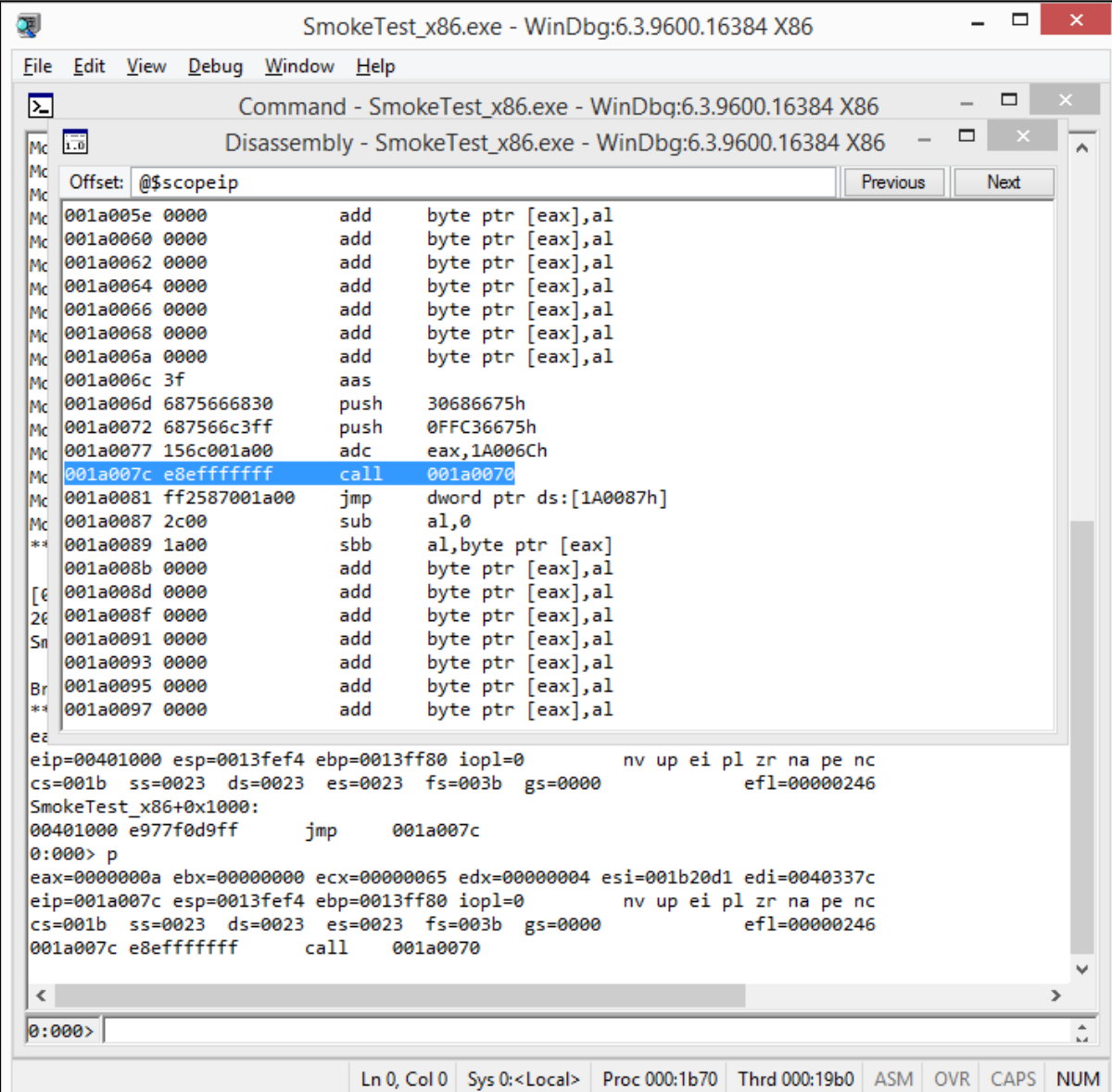
Breakpoint 200001 hit
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\Windows\system32\
eax=0000000a ebx=00000000 ecx=00000065 edx=00000004 esi=001b20d1 edi=0040337c
eip=00401000 esp=0013fef4 ebp=0013ff80 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
SmokeTest_x86+0x1000:
00401000 e977f0d9ff jmp 001a007c

0:000>
```

As you can see, WinDbg first received an external command (via `.ocommand` from Opatch Agent running inside the debuggee) for setting breakpoint #200001 at location `0x00401000`, which is the location where we inject our patchlet code. Subsequently, as execution continued, this breakpoint was hit.

PATCH

Step over the JMP instruction to get into the patchlet code, as shown on the image below. As you can see, the patchlet code consists of a single call (to PIT_ExploitBlocked), after which there is a JMP to the original instructions that were relocated before the Agent overwrote them with the JMP to patchlet code.

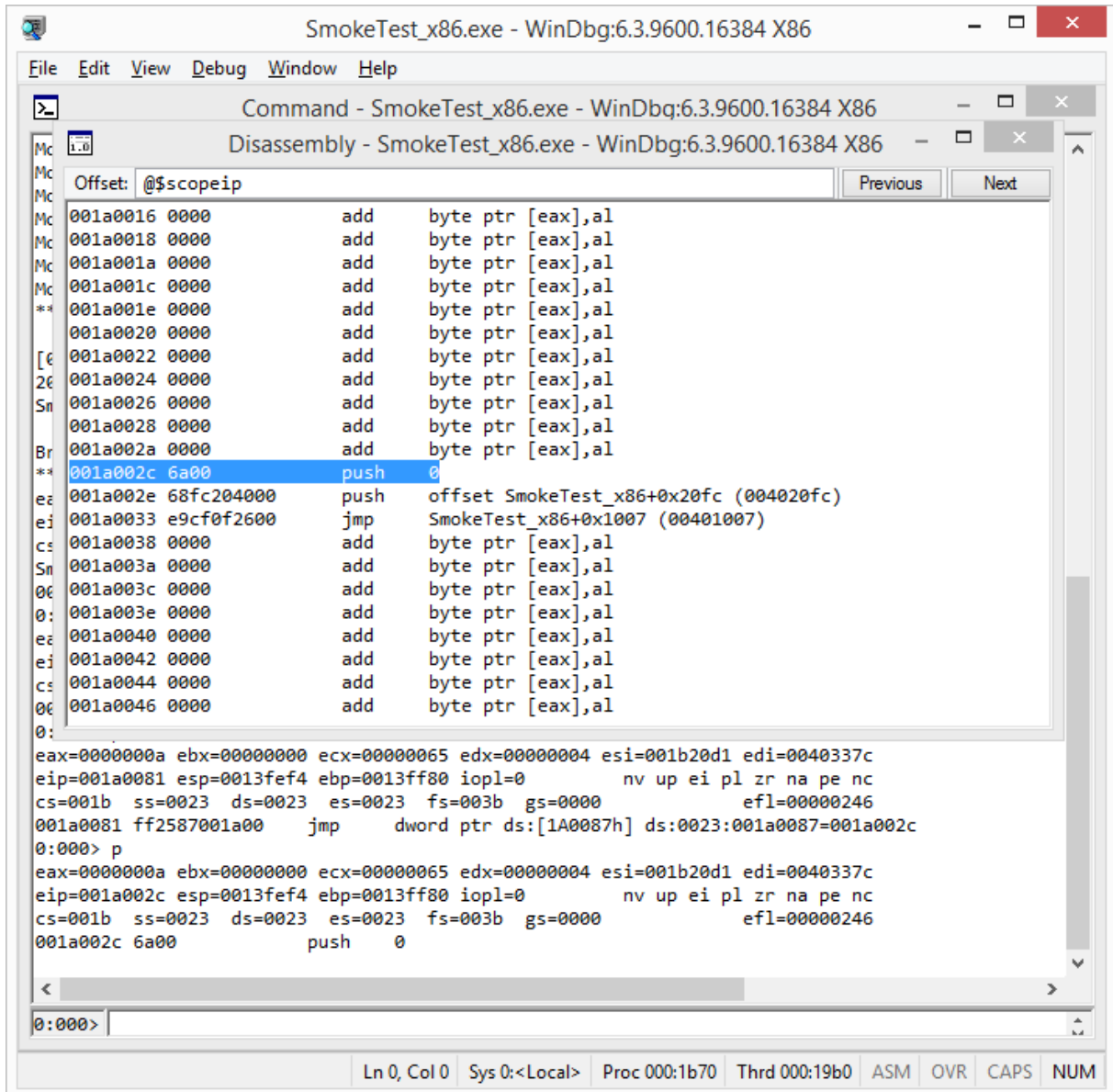


```
SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
File Edit View Debug Window Help
Command - SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
Disassembly - SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
Offset: @$scopeip Previous Next
001a005e 0000 add byte ptr [eax],al
001a0060 0000 add byte ptr [eax],al
001a0062 0000 add byte ptr [eax],al
001a0064 0000 add byte ptr [eax],al
001a0066 0000 add byte ptr [eax],al
001a0068 0000 add byte ptr [eax],al
001a006a 0000 add byte ptr [eax],al
001a006c 3f aas
001a006d 6875666830 push 30686675h
001a0072 687566c3ff push 0FFC36675h
001a0077 156c001a00 adc eax,1A006Ch
001a007c e8efffffff call 001a0070
001a0081 ff2587001a00 jmp dword ptr ds:[1A0087h]
001a0087 2c00 sub al,0
001a0089 1a00 sbb al,byte ptr [eax]
001a008b 0000 add byte ptr [eax],al
001a008d 0000 add byte ptr [eax],al
001a008f 0000 add byte ptr [eax],al
001a0091 0000 add byte ptr [eax],al
001a0093 0000 add byte ptr [eax],al
001a0095 0000 add byte ptr [eax],al
001a0097 0000 add byte ptr [eax],al
eip=00401000 esp=0013fef4 ebp=0013ff80 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
SmokeTest_x86+0x1000:
00401000 e977f0d9ff jmp 001a007c
0:000> p
eax=0000000a ebx=00000000 ecx=00000065 edx=00000004 esi=001b20d1 edi=0040337c
eip=001a007c esp=0013fef4 ebp=0013ff80 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
001a007c e8efffffff call 001a0070
0:000>
```

Step over the CALL to see the “Exploit Attempt Blocked” popup.

PATCH

Finally, step over the `JMP` to see how execution continues with the relocated original code. As you can see on the image below, the original code consists of two `PUSH` instructions; Opatch Agent had to relocate them both as the first one takes up only 2 bytes and the `JMP` to patchlet code needs 5 bytes. After the relocated `PUSH` instructions, you can see a `JMP` to the original code immediately following the original location of these two relocated instructions.



```
SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
File Edit View Debug Window Help
Command - SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
Disassembly - SmokeTest_x86.exe - WinDbg:6.3.9600.16384 X86
Offset: @$scopeip Previous Next
001a0016 0000 add byte ptr [eax],al
001a0018 0000 add byte ptr [eax],al
001a001a 0000 add byte ptr [eax],al
001a001c 0000 add byte ptr [eax],al
** 001a001e 0000 add byte ptr [eax],al
001a0020 0000 add byte ptr [eax],al
[ε 001a0022 0000 add byte ptr [eax],al
2ε 001a0024 0000 add byte ptr [eax],al
Sn 001a0026 0000 add byte ptr [eax],al
001a0028 0000 add byte ptr [eax],al
Br 001a002a 0000 add byte ptr [eax],al
** 001a002c 6a00 push 0
eε 001a002e 68fc204000 push offset SmokeTest_x86+0x20fc (004020fc)
ei 001a0033 e9cf0f2600 jmp SmokeTest_x86+0x1007 (00401007)
cε 001a0038 0000 add byte ptr [eax],al
Sn 001a003a 0000 add byte ptr [eax],al
0ε 001a003c 0000 add byte ptr [eax],al
0: 001a003e 0000 add byte ptr [eax],al
eε 001a0040 0000 add byte ptr [eax],al
ei 001a0042 0000 add byte ptr [eax],al
cε 001a0044 0000 add byte ptr [eax],al
0ε 001a0046 0000 add byte ptr [eax],al
0:
eax=0000000a ebx=00000000 ecx=00000065 edx=00000004 esi=001b20d1 edi=0040337c
eip=001a0081 esp=0013fef4 ebp=0013ff80 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=0000246
001a0081 ff2587001a00 jmp dword ptr ds:[1A0087h] ds:0023:001a0087=001a002c
0:000> p
eax=0000000a ebx=00000000 ecx=00000065 edx=00000004 esi=001b20d1 edi=0040337c
eip=001a002c esp=0013fef4 ebp=0013ff80 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=0000246
001a002c 6a00 push 0
0:000>
```



12. Final Notes

If you're as passionate about fixing vulnerabilities as we are, you will want to get to the point of writing production-quality patches that can actually be distributed to millions of endpoints around the World and potentially applied even before the attackers had enough time to build reliable exploits. We're working hard on building the infrastructure to allow you to submit your patches for an independent quality and security review, and finally have them distributed to users for a decent compensation.

Do let us know about your experience with building patches. We'll appreciate your suggestions, ideas, criticism and bug reports, as well as words of encouragement. We're building this product for you and we need your input to make your work easier and more efficient. Your comments on our documentation are also highly welcome. Email us at support@Opatch.com.

13. Troubleshooting

Problem: Disabling a patch while the patched process is being debugged with WinDbg results in a »Unable to insert breakpoint« WinDbg error.

Solution:

1. This is a bug in our code, waiting to be fixed. Until it is fixed, avoid disabling a patch during debugging.

Problem: Building a Opp file results in a »Buffer out of bounds« error in the last line of the file.

Solution:

1. This is a bug in our code, waiting to be fixed. As a workaround, make sure that your Opp file does not have any empty lines at the end.