How a simple bug in ML compiler could be exploited for backdoors?

Baptiste David
École nationale supérieure d’arts et métiers (ENSAM)
École d’ingénieurs du monde numérique (ESIEA)
Laboratoire de cryptologie et virologie opérationnelles — (C + V)°
38 rue des Docteurs Calmette et Guérin 53000 Laval, France
baptiste[dot]david[at]ensam[dot]fr

ABSTRACT
Whenever a bug occurs in a program, software developers assume that the code is flawed, not the compiler. In fact, if compilers should be correct, they are just normal software with their own bugs. Hard to find, errors in them have significant impact, since it could result to vulnerabilities, especially when they silently miscompile a critical application. Using assembly language to write such software is quite common, especially when time constraint is involved in such program.

This paper exposes a bug found in Microsoft Macro Assembler (ml for short) compiler, developed by Microsoft since 1981. This assembly has the characteristics to get high level-like constructs and high level-like records which help the developer to write assembly code. It is in the management of one of this level-like construct the bug has been found.

This study aims to show how a compiler-bug can be audited and possibly corrected. For application developers, it shows that even old and mature compilers can present bugs. For security researcher, it shows possibilities to hide some unexpected behavior in software with a clear and officially non-bogus code. It highlights opportunities for including stealth backdoors even in open-source software.

Keywords
masm, ml, bug, compiler, macro assembly, backdoor

1. INTRODUCTION
Compilers are piece of software designed to build new software according to their source code. It exists a lot and most famous ones have made most famous software. It exists compilers for every compiled language, such as C#, C++, C and assembly. The last is one of the eldest one, not often used except in critical projects where efficiency and low design consideration must be strongly taken into account.

Write assembly code directly is reputed to be hard and complicated. Every facility provided by more abstract languages are not present and implementing simple algorithms can be a complicated task for developers. There exists different assembly syntaxes (AT&T and Intel for the most famous ones) and different languages, such as GoAsm, NASM or MASM. The last one, MASM [1] which stands for Microsoft Macro Assembler [2], is famous by the ease it provides, since 1981, to implement programs thanks to the use of its macro code [3]. If this ease helps everyday development, this one must be reliable since it rules what programs are supposed to do. In other word, the compiler must not perform any error since these ones are very complicated to find. Indeed, when a bug occurs, the first reflex is about to search from source code and not blaming the compiler. Source code defines the logic the program is supposed to follow. And error coming from a correct source code which would result in an unexpected behavior could be very hard to find. Compilers are supposed to be reliable since software logic depends on them.

Compilers bugs are more common than what is usually thought [4, 5]. A paper [6] about statistics on bugs' compilers examined about five thousand ones on a decade’s span. Technical explanations about such bugs had already been performed for C language [7, 8], for instance. Compilers’ bugs come from implementation errors, undefined behaviour in programming languages or unfortunate optimization made by the compiler [9]. When such errors are found, they must be reported as soon as possible since software can be targeted by unexpected logic problems. But, it could be interesting to imagine how an attacker could exploit such a default. Indeed, open source software are supposed to be secure because everyone can check the source code for unexpected behavior. However an attacker, from a full legitimate open source code, could use a flaw in the compiler to run, at execution time, unexpected features in contradiction of what was originally written in the source code. Different from Unix backdoor [10], this one is close to what is explained in [11]. Hidden from a trap present in the compiler (deliberatly [12] or not), the goal is about to hide the malicious purpose of a program despite its clean source code. Such bugs — designed to build backdoors — can be found with dedicated processes [13] or by the developer when facing an impossible bug in his/her program.

This is the aim of the article since it proposes in section 2 to show and explain a bug in ml compiler — masm compiler coming from Microsoft Visual Studio software. Then, in section 3 would be interesting to explain how an attacker could exploit it. Finally, conclusion about this disclosure will be described.
2. MACRO ASSEMBLY AND ML MIS-MAKE WITH BOOLEAN NEGATION OPERATOR

Assembly language is less complicated than people think. Not far from C language, management of the stack, calling convention and program’s control flow by conditions and loops are main differences from a technical point of view. This is the last point which is interesting in our case. In C, writing conditions is quite simple and the same can be performed with assembly language, as displayed in figure 1.

```
//
// It's a condition.
//
if (a == b && c != 26) {
  // (...)
}
```

(a) Simple C condition

```
mov eax, [ebp+0Ch]; ; Get 'a' value.
cmp [ebp+0Ch], eax; ; Check content of 'b' and 'a'.
jne _out_of_cond; ; Jump out of condition if it's not equal. Otherwise, continue.
cmp [ebp+10h], 26; ; Check content of 'b' with 26.
je _out_of_cond; ; If it doesn't match, go out of condition.
; (...)
; Here comes relevant code when condition is met.
_out_of_cond:
nop
```

(b) Equivalent assembly code.

To understand the assembly code, the reader must take into account that local values are stored on the stack in x86 assembly and, in our example, are referenced from ebp register used as base pointer. Cmp instruction can be viewed as the subtraction between two values with the content of each variable preserved (only the resulting EFLAG register is modified, according to the operation). In case of equality, the difference is zero, otherwise it is not, which explains why jne (jump not equal) instruction is used there.

For obvious reasons, it is more interesting to write algorithms than technical codes, macro assembly such as MASM provides facilities for developers to help them in conditions writing. Something equivalent, close to C language, can be written.

```
mov eax, DWORD PTR [ebp+08h].
.if (eax == DWORD PTR [ebp+0Ch]) && (DWORD PTR [ebp+10h] != 26)
  ; Relevant code when condition is met.
  .endif
```

Figure 2: Macro assembly condition with MASM.

Easier to read, useful for long development, this code is halfway between C and assembly code (note the use of brackets do not change complexity here). Compiled with ml and disassembled with a debugger such as Windbg, the result is provided in the figure 3.

The code is built in order to force the program to jump at 0x010b1045 if one of the condition is not met. Otherwise, the program continues its normal flow to reach 0x010b1044. In addition to AND conditions, MASM can build OR or NOT operators [14]. In the following example, we build two codes to illustrate the action of the NOT operators from macro assembly to generated op-codes by the ml compiler. On the left column, we test whether the value stored in EAX is different from zero or not. If it is not, instruction add ebp, 4 is executed. Otherwise, we go to the following nop instruction. On the right column, the condition is negated, it means if the value stored in EAX is zero, then, the instruction add ebp, 4 is now executed.

```
not eax, [ebp+0Ch] ; Not negate EAX
je 0x010b1045 ; Jump near if equal
```

Figure 3: Disassembly of code in figure 2.

The test operation is performed by or eax, eax1 instruction in both cases. This specific operation could look like a no operation since it is not supposed to perform any modification on EAX content. That is the goal in a sense it preserves the content of the value but set specific flags in EFLAG register to allow conditional instruction to works according to the content of EAX, especially if EAX is zero or not.

The difference comes in the conditional jump instruction which is performed right after. In the first case, je which stands for “jump near if equal” means that jump operation is performed when flag zero is set to one (ZF=1). In other word, if the result of or eax, eax would set ZF to one if EAX was zero. In such a case, the jump will be executed and the instruction provided if condition is met would be skipped (going to nop instruction). Or course, if content of EAX is different from zero, the instructions inside the if statement are executed.

For the second case, the conditional jump is now jne which is the opposite version of je. The condition is met when the zero flag is set to zero (ZF=0). From a logic point of view, it is a negation of the first case. Until that point, everything works correctly as it is supposed to be.
Issues come when we raise the level of complexity in conditions. Consider the following conditions:

```c
.if eax == !ebx
    MOV eax, 2
.endif
.if (!eax) == ebx
    MOV eax, 3
.endif
.if eax == ebx
    MOV eax, 4
.endif
```

**Figure 5:** Use of NOT operator in conditions.

For obvious reasons, all of them are supposed to produce different results, according to the values stored in `eax` and `ebx`. Technically speaking, a condition such as `(eax == !ebx)` could be rewritten as `(eax == (ebx == 0))` to be more understandable. The same applies for `((!eax) == ebx)` and the use or the lack of brackets do not change anything for the compiler. Because all conditions are different, they all should be computed differently (use of different op-codes or conditional jumps, for instance) because each describes a different condition to be met. The main question stands in the following disassembly of these conditions where all conditions are compiled in the exactly same way.

```asm
011b001b 90  nop
011b001c 32  cmp eax, ebx
011b001e 7306 jne main+0x1e (001b0000)  Branch
main+0x001f: 011b0020 8b00000000 mov eax, 2
main+0x0025: 011b0027 75f8 jno main+0x1e (001b0000)  Branch
main+0x0029: 011b0029 8b00000000 mov eax, 3
main+0x002c: 011b002e 7506 jnz main+0x1e (001b0000)  Branch
main+0x002f: 011b0030 7507 jne main+0x1e (001b0000)  Branch
main+0x0032: 011b0032 8b04000000 mov eax, 4
main+0x0037: 011b0037 90  nop
```

**Figure 6:** Disassembly of code compiled in figure 5.

Each condition is built in the same manner (cmp eax, ebx followed by a conditional jump if not equal — jne — to the next condition) despite the fact they are different. From the `ml` compiler point of view, eax == ebx is the same as (!eax) == ebx. Any debugging session is enough to confirm that the logic written originally in the source code is not respected at generation time by the compiler.

To prove an error is made by `ml` compiler, we wrote an equivalent code in C, compiled it with visual studio compiler and reversed it with IDA software¹. The C equivalent code is written with the constraint to retrieve two values from the user. Doing so, no one could object compiler could have made any optimization in the backyard.

```c
int main(int argc, char *argv[]) {
    int a = 0;
    int b = 0;
    // Ask for two numbers to prevent compiler to optimize things with constants.
    tprintf("
    Set number a ");
    scanf("%d", &a);
    tprintf("
    Set number b ");
    scanf("%d", &b);
    // Perform apparently not obvious tests.
    if ((a == b)) {
        tprintf("First line!");
    } else if (a == 1) {
        tprintf("Second line!");
    } else if (a == 2) {
        tprintf("Third line!");
    } else { tprintf("Sorry!");
    return 0;
}
```

**Figure 7:** C code supposed to test NOT condition.

Decompiled code is a piece of cake to check how the compiler built this source code.

Difference with the code compiled with `ml` is blatant. The first block of code is dedicated to retrieve values from the user. Then, the first condition (a == !b) is computed by the use of a temporary value stored on the stack. Indeed, according to the value of b, a temporary local value is set to one or zero (second and third block) to be finally compared to a value at the end. In other words, the negation of b is stored in a `shadow` local value to be used in the comparison next. The same applies for the next condition (in such a case, it is the value a which is used). According to what the C compiler did, there is an issue with `ml` compiler. Note C compiler uses a temporary value

¹https://www.hex-rays.com/products/ida/
to compute the negation of one value, what ml compiler does not. An explanation could sit in the lack of capacity of ml to build its own local values as one developer could craft his/her own ones.

Finally, ml's bug resides in the use of negation operator in an equality (or inequality) test. Every code which follows the pattern !register sign register/memory is eligible to be incorrectly compiled. This bug is introduced by a bad management of the value negated. Indeed, the use of NOT operator changes the content of the value inside the condition. Out of the condition, the negated value is not supposed to be altered. The lack of use of temporary values leads to misinterpret the condition, resulting to the bug. Note that regular negation from a single value (for instance: "if !leax") or a full condition (such as: "if !(condition)") work perfectly fine since it is the test instruction or the conditional jump which is impacted in that case.

3. HOW TO BUILD A SNEAKY BACKDOOR WITH ML COMPILER BUG?

3.1 Context of the backdoor

If it is possible to exploit the bug to execute unexpected code despite official logic of the source code, a backdoor is writable. The goal is to get an official source code which should pass a code review from anyone (expert or not). Of course, reverse engineering compiled code from application would be enough to find the mistake we included in the logic flow of the process... But who cares about reversing open-source application that you can compile by yourself? Of course, because we modify original source code, source code of the targeted application must be accessible. Open-source software are perfect target for that.

As described in [11], two plots are available to build the backdoor. First, you own the project you want to backdoor. On the positive side, it makes it easy to introduce the backdoor since you are the main leader developer. On the negative side, if the backdoor is found, you lose and here comes consequences on your reputation. To reduce the possible cost, the second plot is about to propose a patch for an existing software. It allows you to patch several open-source projects since you can act as an occasional contributor. It is stealthier, in the sense you are not on first line. In addition, you get the benefit to possibly argue it was a plausible mistake instead of a malicious intent by design. In all cases, attack is performed through a modification of an existing software to detour it is original procedure.

The main part, about the modification we submit, is about to not raise a security issue. First, no inspection (from human to full formal verification) of code would be able to detect a security breach in the code since the official security logic is respected. Then, the bug we exploit avoids questions about a dubious (or, at best, a nonsense) functionality inserted in code. After all, we are fooling the logic of conditions in a sneaky way (a condition supposed to be false is going to be interpreted finally, as true) so that code, which is supposed to reject something, now, accepts it.

Even if the code is detected, it is still possible to deny any responsibility in the vulnerability found. The logic of checks in the original code is supposed to be true and you were not supposed to know the debugger had a bug. At worst, one can blame you not to have tested your software enough. Of course, with an example as simple as the one provided in figure 9, it can be complicated to argue we did not test it. But more complex cases can be implemented.

3.2 Description of bug consequences for backdooring

Formally speaking, the bug in the compiler does not take into account the NOT operator in context of equal checks. The legitimate use of NOT operator with equal (or difference) test is with Boolean values (meaning $a \in \mathbb{F}_2$ with $\mathbb{F}_2 = \{0, 1\}$). This is not a big assumption since conditions, where negations are used, manipulate, most of the time, Boolean or equivalent values. With the compiler’s bug, all the following codes are equivalent: $\forall a, b \in \mathbb{F}_2$, we have $a == b \iff !a == b \iff a == !b \iff !a == !b$ which, obviously, leads to issues. A more complex example can be illustrated with: $!a == b)$. In a non bogus environment, we have the equivalent conditions:

$$! (a == b) \iff ! (a \neq b) \iff a == b$$

In the bogus environment of ml compiler, truth equation should change. It is due to the NOT operator which is not interpreted correctly. The first NOT is interpreted on the whole condition while it changes the jump instruction. From an equality, it is now an inequality. The second NOT (in the brackets) is not interpreted at all, which leads to the following conditions:

$$!(a == b) \iff !(a == b) \iff a \neq b$$

With this change in logic of program, we can replace a condition, normally not executed, into one which is now executed. Note the opposite operation is although correct: it is possible to change a condition which is executed to one which is now executed. This is the main point to build the plot of a backdoor.

3.3 Creation of the backdoor

Using logic bug to change behavior of a targeted application is quite simple. We illustrate things with the use of access check to a resource reserved to administrator. For real instances, one can think to programs such as runas or sudo under UNIX subsystem. The goal is to introduce a flaw in a function supposed to check access rights of the calling process before continuing potential privileged functionalities. The simplest example could look like the one provided in figure 9.

With the previous code, if a regular user calls the function, it gains access to the privileged function. This is due to the fact that the NOT operator is not correctly interpreted by the compiler. Finally, ml compiles a condition close to "if caller_context.isAdmin == 1" which triggers the condition, so that the function is stopped. But the code in figure 9 is wrong for an operational backdoor. Indeed, if an administrator calls the function, because of the bug, the legitimate access is now refused. In this case, even the
If the caller is not an administrator, it's an user.

```assembly
    ; Check if the provided user belongs to users
    ; group and save result in esi.
    push offset CurrentUserName
    call IsUserRegularUser
    mov esi, eax

    ; Check if the provided user belongs to
    ; administrator group and save result in edi.
    push offset CurrentUserName
    call IsUserAdministrator
    mov edi, eax

    ; Check password provided with the user name.
    push offset PasswordProvided
    push offset CurrentUserName
    call CheckIfPasswordIsValid

    ; Check if the user is administrator and
    ; the password is correct.
    .if edi == 1 && esi == 0 && eax == 1
        jmp __allow_admin
    .endif

    ; Check for authenticated non administrator
    ; users to perform specific task...
    ; (...)
```

Figure 9: Simple backdoor using the compiler’s bug.

The simplest checking test would found something goes wrong. If we introduce a backdoor, this one must keep the original behavior of the modified code. Inserting a new backdoor functionality does not mean to remove already present functionalities.

The solution comes with a small raise of complexity about the code to backdoor. Continuing the runas example, this one can check, first, if the user belongs to a group, then if the couple user-name and password is correct. At that time, if the user belongs to the group of administrator, access can be guaranteed. A correct implementation would lead to check all these operations ones after the others. But, for optimizing things up, we propose to check if the provided user-name is administrator or just a regular user (implicitly supposing there is no other group except administrator or regular user). Finally, the check of provided password can be performed. The resulting code can be seen as the following.

Figure 10: Check to let access to provided login/password.

The double check of the user affiliation can be justified by the fact that non administrator user could perform some task in the coming code or something else, close to that explanation. The code in figure 10 is the original code to backdoor. Of course, the main working space is in the condition. First, we are going to study the condition with a truth table, according to all possible outputs of values:

| Administrator | User | Password correct | Result |
|---------------|------|------------------|--------|
| 0             | 0    | 0                | 0      |
| 0             | 0    | 1                | 0      |
| 0             | 1    | 0                | 0      |
| 1             | 0    | 0                | 0      |
| 1             | 0    | 1                | 1      |
| 1             | 1    | 0                | ?      |
| 1             | 1    | 1                | ?      |

Note the last two possibilities should never happen. Indeed, by definition, a user cannot be both administrator and regular user (ie: non administrator) and even if it was possible, there is no real definitive answers to provide here. In a way, these undefined states could be an interesting stop to exploit our compiler’s bug... Technically, our goal is to get a regular user (ie: non administrator), with the correct password of the user account, being executed as an administrator, in addition to administrator users whose execution is still guaranteed. First, we are going to modify the condition in a manner it does not change the logic of the condition but it introduce our NOT operator. One can write:

```assembly
    .if edi == 1 && !esi == edi && eax == 1
        mov ebx, 1
        jmp __allow_admin
    .endif
```

Figure 11: Change the condition (but not the logic) to introduce NOT.

The code in figure 11 introduces the NOT operator but it does not change the logic of the condition from a formal point of view. Note that there is many ways, with the compiler’s bug, to introduce this operator (for instance, !esi == edi since edi is supposed to be at one). This introduction is targeted on the esi register since this one is supposed to contain the Boolean value defining the belonging of the provided user to the group of regular users. This is in order to force the misinterpretation the real membership of the provided user at execution time.

The logic of the condition is to only accept what is defined as true to let privileged functionalities being executed. With just the last modification, no user (administrator or not) would be executed. Indeed, the condition is now only valid if the user is identified as being membership of both administrator and non administrator group. This is due to the use of AND operators in the condition. To remove this obstacle, we can define condition in other order and by the use of negative logic. Everything which is identified not to be an authenticated user will be rejected, allowing the execution to continue otherwise. Of course, because the undefined state is not supposed to happen in theory, we do not implement it in our code. A possible solution can be
seen in code written in figure 12.

```
.if (eax == 1 && (!esi == 1 || edi == 1))
    jmp __allow_admin
.else
    jmp __fail_access
.endif
```

Figure 12: Trapped condition.

Condition in figure 12 is conform with the logic of the original condition in figure 10. It first checks whether the password is correct or not. If this one is not compliant, the code jumps directly to the failure part of the function. The justification of this first check lies in the need to avoid unnecessary checks if the password is invalid. Then, if the password is correct, the condition checks if the provided user is either an administrator or a regular user. This double check is to officially avoid undefined membership status of users...

The reality of the compiler’s bug forces the condition to be interpreted differently. Actually, the sub-condition `(esi == 1)` is interpreted as `(esi == 1)`. Under this condition, it allows an administrator or a regular user, provided the password linked to its account, to get access to privileged functionalities. The following array resumes the truth table really compiled by the ml.

| Password correct | User     | Administrator | Result |
|------------------|----------|---------------|--------|
| 0                | 0        | 0             | 0      |
| 0                | 0        | 1             | 0      |
| 0                | 1        | 0             | 0      |
| 0                | 1        | 1             | 1      |
| 1                | 0        | 0             | 0      |
| 1                | 0        | 1             | 1      |
| 1                | 1        | 0             | 1      |
| 1                | 1        | 1             | 1      |

The antepenultimate and the penultimate lines define the expected result. Either an administrator or a regular user — correctly authenticated — checks the condition. This is the design of the backdoor we introduce. It allows one more hidden case to get access to privileged functionalities while keeping the original behavior from the condition. Remaining the case of the last line, this one is not really relevant, since it is theoretically not expected a single user can be a member of two groups at the same time. And, even if it could happen, officially, the condition would give access if there is admin rights. To some degree, condition made a choice to privilege admin users, whatever they belong to another group at the same time... A perfectly sustainable choice.

At the difference of [11], our code does not appear as needlessly complicated. It is perfectly justifiable without betray its unexpected design. One malicious author could insert such a code as a patch and justify it by optimization purposes, as we did in this paper. Another difference lies in the flexibility of our backdoor. It can be introduced in any condition, changing the logic flow of this one without removing the original behavior of the condition. To perform the modification, one can follow these steps:

1. Define the original truth table of the condition;
2. Write the expected truth table for the new condition;
3. Build a legitimate condition which follows the last truth table by the use of NOT operator in an equal test condition to exploit the bug.

4. CORRECTION OF THE BUG

Once the bug has been discovered, and because it is potentially critical, it has been transferred to Microsoft under CVE number CVE-2018-8232. Since the bug is clearly identified, detection of specific patterns in conditions is enough to detect potential issues. From that point, many possibilities are doable. First, the compiler could display warning or error messages to prevent compilation of such code. If it does not fix the problem, it allows developers to be informed about it and to avoid such a dangerous use. The second possibility is to increase the volume of generated code for such configuration. As presented in Section 2, the main problem comes from the lack of a temporary variable. These two solutions are presented in the current section.

In addition, it was required to inform people about the necessity to review all their source-codes which used that compiler. To correct the bug, many solutions exist to do it. On the first hand, it would be about adding a warning when the compiler encounters such a construction (detected as previously bugger). This warning informs a potential flow in the construction when the negation must be performed with a temporary variable. On the second hand, the compiler can insert, by itself, this temporary variable to correct the logic. With the compiler’s bug, we are dealing with incorrect code generated in the following situation.

Correction can be designed in the same way the C compiler did it in figure 8. By saving original value of the value or by creating a new one, the NOT operation can be performed (explicitly by adding a test or just by using dedicated instructions). In that way, the compiled code could be generically corrected to something close to the code displayed in figure 14.

The main idea is to save the negated value in the stack to use it directly in the original condition. Using stack is not a big issue since this is the regular procedure to create...
local values for functions. Note this temporary value used must be removed from the stack whatever the result of the condition was. It is in order to keep coherency for previous values already present in the stack.

The decompiled version of the previous code follows requirement of the condition originally defined by the developer. The complexity cost is relatively small since operation on stack are common. No double code displayed in figures 14 et 15 could be improved to something more efficient.

Figure 14: Proposition of solution to correct the bug.

The original test by itself.

```Assembly
; Test first to negate eax.
.if eax == 0
    ; Use stack as temporary value.
    push 1
.else
    ; If eax is 1 it becomes 0.
    push 0
.endif

; Clean the stack, whatever the result
; of the test is.
add esp, 4
xor eax, eax
.endif
```

Figure 15: Decompiled correction for the proposed solution.

More than correcting the bug in the compiler by Microsoft, a large audit of applications developed with that compiler is mandatory. Indeed, a compiler is a software responsible to generate other software. It means that other source codes have been compiled with it and these ones should be audited in order to find bogus constructions in conditions, on the first hand, and to confirm the original behavior of the software once this one has been recompiled (since condition flow could change) on the other hand. Taking into account that assembly language is used by some firmware builder, drivers developers or just for tenuous part of critical code, it can be complicated to update these software since some code can potentially not be updatable (lake of internet connection, code written in ROM for embedded devices, small portion of code used in a math library which is part of a bigger project...). Including the fact that ml is an old compiler, it represents a large number of software which have been made with it by a lot of people. Even if a CVE number has been assigned and a security update pushed, it could be hard to fix everything, especially if the source code of one software has been developed at former time by a company which has collapsed since that time. In such a case, a reverse engineering process should be made on compiled code to check that logic flow for each condition is designed to perform the task for those it has been designed.

5. CONCLUSION

This bug has been found during the development of specific processes dedicated to security. If there have always been debates over the usefulness and performance of assembly language relative to high-level languages, this one is still used in industry. Far to be the most common, the TIOBE index of programming language popularity ranks assembly language at 15 in February 2018 and its use is growing these last years. Just from open-source world, one can find easily more than one thousandth projects using MASM in their source code. We find assembly developers in the fields of compiler, firmware, security, real-time simulations or calculation and anti-malware engineering. All are critical fields and where programs run with high privileges and full access to user’s data.

After numerous exchanges with Microsoft team, a solution has been implemented to fix the bug. According to the previous section, several possibilities could be performed, including rejecting the compilation with an error or building the binary as C compiler does. But, because assembly language must give to the developer a full control of what it is compiled and in order to not break backward compatibility with existing software, Microsoft decided to correct the bug with the error message. Now, A2154 error with the message “syntax error in control-flow directive” is displayed when such construction in condition is used. From an operational point of view, recompiling existing codes with the new version of Visual Studio 2017 (July 2018) is enough to detect a potential use of the vulnerability.

More generally, using a bug in a compiler to introduce a trap in a software is a nice and smooth manner to get access to a targeted system in the stealthiest way. From a specific compiled version dedicated to a specific target to a wildly used open-source software, possibilities are endless. Undetectable for humans, perfectly justifiable for malicious developers, thin and efficient, this type of backdoor is designed for long term and high efficiency. Detectable by automatic tests process, these one must be calibrated to record and take care of suspicious signal or

---

3 https://www.tiobe.com/tiobe-index/
4 https://www.tiobe.com/tiobe-index/assembly-language/
5 https://searchcode.com/?q=masm&lan=34
undefined states. Of course, example provided in this paper is for illustration purpose and it could be possible to detect it with dedicated tests. But, with no prior knowledge a backdoor has been inserted, detection by specific test procedure can be almost impossible to perform. Note that, even if tests find something tendentious, auditing source code would not be enough to fix the problem and it would be even worth to identify the bug as a malicious trap inserted on purpose... Reverse engineering is the only solution to understand and fix bug (both from the targeted software where the trap has been set and the bugged compiler), but it is not a mass sport and not a basic skill earned by each developer. Note that it requires that developers suspect bugs come from compiler and not their own code to first be able to fix things. A state of mind which is far to be common if you are not aware it could be possible.

From an operation point of view, such a backdoor is almost perfect. Especially with the bug provided here, attacker has the ability to insert new possibilities in any critical condition targeted without modifying original behavior of the condition. Bonus, attacker can stop diffusion of the attack by correcting bug in the compiler, removing the trap for new compiled version without changing targeted software’s source code. Another bonus lies in the difficulty to update critical pieces of code, most of the time in firmware and other close to hardware or kernel components. Last attacks such as Spectre [15] and Meltdown [16] demonstrate how hard it could be to change firmware in critical pieces of software. Not using such extreme examples, thinking about pieces of software in firmware, written in assembly, sometime years ago, by companies which could not exist any more, could make impact of such backdoor very critical.

Of course, the goal of this paper is to report this vulnerabilities to make it patched as soon as possible. But it aims to explain how to find such a bug, make it visible to any developer and alert about how important damage could be if it was exploited by malicious developers. More important, it underlines the necessity to not believe blindly that open-source projects are secure. Even if they are correctly audited by experts. In the same philosophy than [11], our work is different since it does not require to use obfuscation to hide the trap, which make our perfectly justifiable. Compiler is a milestone in the building procedure of an application and it must be not underestimated as a potential source of vulnerability. This is not new since Ken Thompson [12] did it in 1984, by modifying the compiler and then compiling a trapped code. Here, we exploit a zero-day vulnerability in the compiler to do the job.

6. REFERENCES
[1] Colin Robertson. Microsoft macro assembler reference. May 2016.
[2] Watt Peggy and Christine McGeever. The ibm pc macro assembler was released in december 1981. 7, January 1985.
[3] Marshall Martin. Macro assembler update adds high-level features. 13, April 1991.
[4] Attila Szegedi, Adam Berger, and Dave Smith. Compiler bug, a semi-mythical beastie. March 2010.
[5] John Regehr. Is that a compiler bug? February 2010.
[6] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. ISSTA ’16, 2016.
[7] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), June 2011.
[8] Chengnian Sun and Vuand Zhendong Le, Su. Finding and analyzing compiler warning defects. ICSE ’16, 2016.
[9] Eric Raymond. When you see a heisenbug in c, suspect your compiler’s optimizer. February 2010.
[10] Ed Felten. The linux backdoor attempt of 2003. October 2013.
[11] Scotty Bauer, Pascal Cuoq, and John Regehr. Deniable backdoors using compiler bugs. International Journal of PoC|GTFO 0x08, June 2015.
[12] Ken Thompson. Reflections on trusting trust. Commun. ACM, 27(8):761–763, August 1984.
[13] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. OOPSLA ’15, June 2011.
[14] Microsoft Corporation. Programmer’s guide: Microsoft masm : Assembly-language development system : Version 6.1 : Form ms-dos and windows operating systems. Microsoft Corporation, pages 141–142, 1992.
[15] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. ArXiv e-prints, jan 2018.
[16] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. ArXiv e-prints, jan 2018.