UQBTng: a tool capable of automatically finding integer overflows in Win32 binaries

Rafal Wojtczuk
Warsaw University
rafal.wojtczuk@mimuw.edu.pl

November 27, 2005

***Abstract*— This paper outlines the recent work by the author to develop UQBTng, a tool capable of automatic detection of exploitable integer overflow bugs in Win32 binaries. A brief description of this programming error is given, along with rationale for focusing on Win32 assembly security checking. The tool heavily relies on the excellent UQBT[1] package; the description of the applied enhancements to UQBT is presented. Although definitely a work in progress, UQBTng already can produce results helpful for a security researcher; final section includes the analysis of the run of the tool against a binary (implementing a certain service in Windows 2000) with known integer overflows.**

***Index Terms*— computer security, integer overflow, Windows, UQBT, program verification.**

I. Introduction

THERE is a plethora of academic papers on verification of software. Unfortunately, their usefulness for a security practician dealing with common flaws in popular operating systems and applications [2] is usually very limited, for variety of reasons. Some verification methods require a lot of effort (and knowledge) from the user to be put into formal proof of correctness, using general purpose theorem provers. Other methods, most notably ones related to model checking, require to build a model of a system to be proved, which is both labor-intensive and error prone. Finally, many papers describe methods which sound appealing, but which are applicable only to small programs. Therefore academic papers are seldom posted to or discussed on mainstream security mailing lists [3].

There are notable exceptions though. One particularly promising way is to give up trying to prove full correctness of a program (let's name it "verification"), as the associated cost is high. Instead one may attempt to check certain properties of a program, which are known to cause security problems[1]. Ideally such properties should be local and in order to check them, one does not need to know full semantics of an analyzed program. Let's name this approach "checking"; a good example, with real security vulnerabilities discovered, can be found in [4].

Current popular OSes are usually written in C and C++. If there is no source available, one is faced with analyzing the resultant assembly code. While there are a number of papers related to verification or checking of C code, very few tools exist which are capable of advanced reasoning about compiled C code[2] . However, this is an important issue. Particularly, MS Windows operating systems family is of great interest to security researchers, due to its popularity, long history of known security problems, and probably high number of still undiscovered flaws [5].

For all the reasons mentioned above, the author decided to invest some effort to create a tool capable of searching the Win32 assembly code for a particular vulnerability - the integer overflow bug. The tool is meant to require as little interaction from an user as possible - average size of a program or library on a typical Win32 system is measured in hundreds of kilobytes, and we cannot afford to annotate or otherwise specify semantics of too many points in the code.

II. The integer overflow vulnerability

The title vulnerability is the cause of many recent serious security problems in many popular operating systems, even in the components designed to be secure[6]. Particularly, the Microsoft security bulletin MS05-053[10] describes a vulnerability in the GDI32.DLL library, which was remedied by adding over 50 integer overflow checks to this library .

The nature of this type of vulnerability is simple - due to the limited range of numbers which can be stored in a C language integer variable, it is possible that during arithmetical operations (most often addition or multiplication) the value of the variable may silently overflow and wrap, and become smaller than the sum (or product) of the operands. If the result is used as the size of memory allocation, then subsequently a buffer overflow can occur, which may yield the attacker full control over the code execution. Example:

```
void* vuln_func(void* data, unsigned int
len)
{
    unsigned int size=len+1;
    char* buf=malloc(size);
    if (!buf) return NULL;
    memcpy(buf, data, len);
    buf[len]=0;
    return buf;
}
```

This function copies user-supplied data into a new buffer and null-terminates it. If an attacker can pass 0xffffffff as

[1]Observe such an approach is often a basis of a security audit of software

[2]The "binaudit" tool, announced on www.sabre-security.com site, could be a very interesting piece of code (judging at least by the reputation of its author), yet for a long time it is still in development, and little details are known on its internals

the ”len” parameter, an integer overflow will happen when calculating ”size” variable; malloc will allocate a memory block of size 0, and the subsequent memcpy will overwrite heap memory.

This vulnerability has an important feature - usually a prover/checker does not have to understand loops semantic to detect this bug; simple arithmetics should suffice. It is well known that reasoning about loops is difficult[3] - usually it is required to manually provide a loop invariant, which is not trivial. Therefore, if we focus on detecting integer overflows in memory size calculations, we have a good chance of succeeding with automated checking. Also, this vulnerability is usually quite local, in the sense that the size calculation (which should include a check for integer overflow) is usually close to the actual memory allocation, which makes it possible to reliably detect the presence or lack of the vulnerability by analyzing a single function or a small number of functions.

III. Necessary components

A. Summary

In order to reliably reason about the assembly code, two components are needed:

- a decompiler capable of decoding single assembly instructions into a form with easily accessible semantics, as well as recovering higher level C code constructions; it is not necessary to produce a real C code, but as we will see we will choose this way
- theorem prover or model checker, capable of reasoning about the code semantics

As we will see, we will take a (modified and enhanced) decompiler, add functionality to automatically annotate the decompiled code with assertions which check for integer overflows, and then feed the decompiled code into the checker to verify the existence of integer overflows.

Two excellent, publicly available academic tools: UQBT[1] and CBMC[7] were chosen for this task[4]; they are briefly described below.

B. CBMC: Bounded Model Checking for ANSI-C

CBMC[7] is a checker capable of automatically proving properties of ANSI C code. The properties are embedded in the checked code in a form of usual C ”assert” statements. CBMC is unique in its ability to support almost all features of C language; particularly, the following constructions are handled well (while other C code checkers usually have problems with them):

- arithmetics with bounded integers
- pointer arithmetics
- bitwise operations
- function pointers

CBMC works best on a code without loops. When a loop is present, CBMC can be instructed to unwind it a couple of times and possibly verify that a given number of unwind operations is sufficient; however, in most cases, it is not enough. But as noted above, integer overflows are usually not caused by calculations implemented by a loop. Therefore, for the purpose of checking for integer overflows only, we can remove all looping constructions from the analyzed code, without significant loss of functionality. In such case, checking with CBMC is fully automatic; if an assertion does not hold, an appropriate counterexample is presented to the user.

C. UQBT: University of Queensland Binary Translator

UQBT[1] is an executable translator - it takes as input a binary file from an operating system A, decompiles it, and then it can produce a binary for a different system B, preserving the binary semantics.

For our purposes, the most important capability of UQBT is its ability to decompile an executable into a graph of functions. Each instruction in a function is represented by a ”semantic string”, a data structure capturing the semantics of an instruction. A lot of code is available to process semantic strings; for example, there is a function which can replace each occurrence of a subexpression (say, a dereference of a location pointed by a frame pointer minus a constant offset) in a semantic string with another expression (say, a semantic string describing a local variable). Therefore it is possible to process the instructions effectively and conveniently.

Another tool was considered for the task of decompilation: The Interactive Disassembler[8]. However, IDA has numerous disadvantages:

- The advanced functionality is not documented (besides sparse and insufficient comments in the header files).
- Probably[5] the internal instruction representation is too close to the actual assembly; on the other hand, UQBT uses quite high level, portable representation.
- There is no source available; it is a non-free software.

The above issues (and a few others, less important) decided against IDA. Admittedly, IDA has some advantages; it is a very stable software and its accuracy in some aspects is very appealing (particularly, most of the PE file format analysis functionality implemented by the author so far for UQBTng is already present in IDA). However, in the long run UQBT should be the better choice.

UQBT is a large piece of software; it can handle Pentium and Sparc architecture, and recognizes ELF, MSDOS .exe and (to some extent) Windows PE file formats. A useful feature is its ability to produce a C code from a binary; though not necessary for analysis (actually, analysis is done on the semantic strings, not on the C code), this makes it easy to produce input which a theorem prover or checker can work on.

For our needs, the ability to decompile Win32 PE files is crucial. Functionality of UQBT had to be enhanced to process PE input files more accurately. The next section describes the most important modifications.

[3]Yes, undecidability. Each good paper should include this word at least once.

[4]UQBTng uses a development version of CBMC, provided by its author

[5]The author tried to assess some aspects of IDA advanced functionality, however it was difficult due to the lack of the documentation; therefore this description may be not 100% accurate

IV. UQBT MODIFICATIONS

A. Summary

Among all the binary file formats supported by UQBT, the ELF format is handled most exhaustively. In case of PE file format (the default format for executables and libraries on Windows OS), significant enhancements had to be added in order to capture the semantics of the code. Some of them are related to peculiarities of the compiler; other are forced by CBMC properties. The most important code additions are enumerated below.

B. Library functions

Certain library functions (for example `LocalAlloc`, `wcslen`) are crucial in the assertion generation algorithm (described in the following section). As UQBT did not recognize the library function usage in PE files, appropriate support had to be added.

In order to get the list of the imported functions, it is enough to locate in the PE file the data structure named "import lookup table" and parse it (see [11]). For each library function F, in the address space of the Win32 process there exists a 32bit location (an import address table[6] element) which is filled with the address of F by the library loader. The library function can be called in three different ways:

1) Direct call of the address stored in IAT entry: `call ds:iat_entry`
2) Call to a "thunk" function, which consists of a single jump instruction: `jmp ds:iat_entry`
3) Assignment to a register, then call register: `mov ebx,ds:iat_entry; call ebx` This convention saves space when more that one call to the same library function is made subsequently.

In the first two cases, it is easy to determine whether a given instruction is in fact a library function call: just check whether the argument to the "call" or "jmp" instruction is within import address table range. However, because of the third case, for each "call register" instruction, we have to find the instruction X which assigns the register. The instruction X can be quite far away from the place where the actual function call takes place. Therefore, a reliable algorithm to trace back the execution flow had to be implemented; particularly, jumps and conditional jumps have to be back-traced.

C. Calling conventions

The calling convention describes how parameters and return values are arranged for a function call. In case of ia32 architecture, the most commonly used convention (named "cdecl") is:

1) Parameters are passed on the stack
2) Parameters are removed from the stack by the caller

UQBT supports only the above convention. However, Win32 binaries use the following conventions:

Conv name	Args passed...	Who removes args
cdecl	on the stack	caller
stdcall	on the stack	callee
thiscall	first arg in register ecx, the rest on the stack	callee
fastcall	first two args in ecx, edx, the rest on the stack	callee

"Fastcall" convention is used very rarely and can be ignored. "Thiscall" convention is used for passing "this" parameter to a class function; as currently we do not handle object code well for many other reasons[7], we choose to ignore it for now as well.

This leaves us with the problem of distinguishing between cdecl and stdcall functions. The failure to do it properly results in incorrect view of the stack after the function has returned; it is particularly damaging when the analyzed procedure has not set a frame pointer.

If a called function is implemented in the code we are analyzing, it is easy to find out its convention type: if the return from the function is implemented by a `ret` instruction, then it is a cdecl function; if `retn N` instruction is used, then it is a stdcall function, and its arguments occupy N bytes. However, in case of a library function, this method obviously does not work.

The following solutions to the problem were considered:

1) Retrieve the calling convention information from header .h files shipped in WINDDK. Disadvantage: many Win32 library functions are undocumented (in header files or anywhere else).
2) Retrieve the calling convention from the debugging symbol file (.pdb). Imported functions honoring stdcall convention are represented as `name@N`, where `name` is the function name, and `N` is the amount of space occupied by the arguments. Disadvantage: usually debugging symbols are not available (Windows OS binaries are an exception to this rule), so relying on them would limit the applications of the tool.
3) Assume that functions imported from `MSVCRT.DLL` use `cdecl` convention, and other functions use `stdcall`. Detect exceptions to this rule by observing "stack type assumption violated" error messages in the logs, and then manually annotate offending functions. Disadvantage: for each analyzed binary, a few functions must be manually annotated.

The last option was chosen, as it provides maximum flexibility with acceptable manual labor overhead. Nontrivial amount of code was written in order to determine the amount of parameters passed to each call to library function, as well to fix the stack pointer after the stdcall function return.

D. Function prologue and epilogue patterns

UQBT assumed that instructions which constitute a function prologue (or epilogue) are not intermixed with other

[6] IAT for short

[7] see the list of possible extensions in the last chapter

instructions. This assumption does not hold in case of binaries compiled with Visual C compiler; particularly ebx, esi and edi register saving is often delayed. This sometimes created a condition where registers save and restore were not paired, resulting in stack height inconsistencies. In order to solve this problem, register save/restore instructions were disassociated from prologue/epilogue patterns, and now they are decoded generically.

E. Handling of "finally" functions

The "finally"[8] construction is implemented by Visual C by creating functions which exhibit two anomalies:

1) They access the caller frame (do not save or set ebp register, and access unmodified ebp register)
2) Sometimes they perform unwind operation, returning directly into its caller's callee.

These functions are detected by examining the Structured Exception Handling setup code and extracting the appropriate pointers. Currently processing of such functions is disabled.

F. Register overlap handling

In ia32 architecture, there are instructions which operate on 8bit or 16bit parts of 32bit registers. In the C code generated by UQBT this feature is handled by defining each 32bit register as a union, consisting of a single 32bit location, two 16bit locations and four eight bit locations:

```
union  {
     int32  i24;
     struct  {
         int16  h0;
         int16  dummy1;
     } h;
     struct  {
         int8  b8;
         int8  b12;
         int8  dummy2;
         int8  dummy3;
     } b;
} i24;
```

32bit register eax is modeled by i24.i24, 16bit register ax is modelled by i24.h.h0, and 8bit registers al and ah are modelled by i24.b.b8 and i24.b.b12, respectively. Obviously, any modification to e.g eax model automatically results in modification to ax model.

Unfortunately, CBMC forbids access to a union member if the last operation on the union modified other member, the reason being that semantics of such operations is endianness-dependent and should be avoided.

The solution is to abandon the above union trick and declare separate storage for each register. To minimize the code changes, it is enough to change the top "union" keyword in the previous code fragment to "struct". Then we treat the 32bit register as the primary one, and we will update registers before or after the assignment:

- before assignment - if a smaller register is in RHS, update this smaller register content with appropriate portion of the 32bit register
- after assignment - if a smaller register is in LHS, update the appropriate portion of the 32bit register with this smaller register

For instance, the instruction and al, 2 will be translated to

```
/* 8bit regs in RHS update:*/
i24.b.b8  = ((unsigned  int)i24.i24)%0x100;
/* the original assignment */
i24.b.b8  = ((int32)i24.b.b8)&(2);
/* 8bit reg LHS update */
i24.i24  -= ((unsigned  int)i24.i24)%0x100;
i24.i24  += (unsigned  int)i24.b.b8;
```

The rational assumption is that non-32bit operations are much less frequent than the 32bit operations, therefore the number of the above updates will be a fraction of the number of assignments. In the analyzed case of the NWWKS.DLL binary, out of 17049 generated assignments 394 were the ones related to overlapping registers handling.

G. Inlined, optimized common functions

In a compiled C code, probably all occurrences of the Pentium instructions with "rep" prefix are generated by inlining an optimized version of one of the following functions:

- strlen
- memcpy
- memset

It is beneficial to replace code fragments implementing above functions with calls to an appropriate function. UQBT includes a few patterns of such constructions, however they did not match the code generated by VC compiler. Appropriate support has been added.

V. Adding checks for integer overflow in memory allocation

The following algorithm was used to generate assertions which check for integer overflow in memory allocation:

- locate all occurrences of calls to functions which allocate memory; LocalAlloc and GlobalAlloc functions are handled by default, other memory allocation functions can be specified in a config file
- execute find_and_annotate algorithm with arguments: the function actual parameter determining the size of allocated memory, and the address of the code where the function is called

The algorithm find_and_annotate(sem_str, code_address) performs the following steps:

- if the sem_str is a constant, exit
- starting at the instruction with the address code_address in the control flow graph, trace back through all execution paths looking for an assignment *A* whose left hand side is related (see the next two points) to sem_str

[8] a part of try-finally construction used with exceptions

- if LHS of *A* is exactly sem_str, then precede *A* with an assertion checking whether integer overflow can happen in *A*; for instance, for a 32bit addition
  v1 = v2 + v3
  generate an assertion
  assert((unsigned)v2 $\leq$ 4294967295 - (unsigned)v3)
  similarly for multiplication with a constant.
- if sem_str is a register *Rshort* and LHS of *A* is a register *Rlong* which is wider than and overlaps *Rshort*, then place after *A* an assertion checking whether the value of *Rlong* is not larger than the maximum value of the type of *Rshort*
- for each subexpression *S* of the right hand side of *A*, execute find_and_annotate(S, address_of_A)

As the above algorithm traverses a (possibly cyclic) graph, care must have been taken to avoid infinite looping.

Due to the complexity if the problem, no action is taken to detect pointer usage condition, so in the following example:

```
varptr=&lenvar
lenvar=eax;
*varptr+=16;
LocalAlloc(heapdesc,        lenvar);
```

the addition instruction will not be annotated. It is believed that intermixing operations on a variable and operations on a pointer to the same variable should be very rare in a C compiled code.

VI. Preliminary results

A. Summary

As stated above, UQBTng is in an early alpha state. For most binaries, it will not produce satisfactory results due to inability to follow pointers usage. However, a test case is available which demonstrates the current capabilities of the tool.

B. The test target

Microsoft Security Bulletin MS05-046 titled "Vulnerability in the Client Service for NetWare Could Allow Remote Code Execution" describes a security hole in one of OS services. Successful exploitation of this vulnerability enables an attacker to take full control of the affected system. This service is implemented by a library nwwks.dll. UQBTng was run with this library as an input.

C. Additional specification for UQBT

In order to obtain better coverage of the code, the list of all functions along with their addresses were retrieved from the debugging symbols of nwwks.dll library and made available to UQBTng.

The first run of the tool produced a couple of "stack type assumption violated" errors. Manual inspection of the code fragments[9] referenced in the above errors quickly determined the three library functions which apparently not did obey the default "stdcall" calling convention, namely

- imp_DbgPrint
- imp_NwlibMakeNcp
- imp_wsprintfW

These functions were added to the config file common.h and marked as "cdecl" functions. The next run of the tool finished without errors.

D. Additional specification for CBMC

The first run of the checker produced over 20 alerts about violated assertions. The analysis of the first case quickly discovered the reason: the respective code looked similar to this example:

```
eax  = imp_wcslen(somestring);
eax  = eax*2+2;
eax  = LocalAlloc(heapdesc,        eax)
```

As we see, the length of an Unicode string is calculated and the appropriate amount of memory is allocated (probably for future copy operation). As the imp_wcslen function measures the amount of memory occupied by a string (by searching for a terminating two null bytes), it is not possible to cause integer overflow in the above calculation[10]. Moreover, it makes sense to assume that the string length is limited by, say, RPC runtime.

Therefore, the following function definition was added to the set of functions produced by UQBTng:

```
unsigned   int  imp_wcslen(arg)
{
return   nondet _uint()%16000000;
}
```

thus effectively informing the checker that maximum value returned by _imp_wcslen is limited.

After this addition (and analogous for function imp_strlen) the next run of the checker returned seven failed assertions; they are briefly analyzed below.

E. Seven failed assertions

Among the seven failed assertions, three were caused by real bugs; exploitation of each of these bugs enables an attacker to perform a heap overwrite and gain the control over execution of the service.

One false positive was caused by the lack of information about the semantics of the library function imp_RtlInitUnicodeString.

Two more false positives were caused by unsupported pointer operations.

The last false positive (in function proc57) was caused by the fact that currently CBMC is instructed to check each generated function separately. When CBMC was passed as arguments the file proc57.c along with the files containing the callers of proc57(), namely proc12.c and proc30.c, the checking succeeded. It is a straightforward task to create

[9] As usual, IDA was indispensable for manual code analysis

[10] On the contrary, let's assume that a certain string representation data structure stores the string length explicitly in a field X, and a malicious party may craft a structure whose actual size may not be equal X. The data structure BSTR, used in COM, is an example. Then, if a function analogous to imp_wcslen simply returned contents of the field X, integer overflow in a calculation analogous to the above would be possible.

scripts which will check each function separately, then check each function together with its callees and callers, etc etc. The problem is that CBMC sometimes requires huge amount of RAM even to process a single small function; some work is needed to minimize the memory requirements. Until this issue has been resolved, the implementation of checking multiple functions simultaneously is deferred.

F. Statistics

The target nwwks.dll binary has size 60688 bytes.

On a machine with Pentium 4 2.4 GHz processor, UQBT generated 215 functions, totaling 661946 bytes of C code, in 20 seconds; maximum RAM usage reached 112 MB. The calling convention of 3 library functions had to be specified manually in order to finish decompilation without errors.

For 60 LocalAlloc invocations, 132 assertions were generated; simplified semantics of two library functions had to be specified in order to check most of the assertions successfully. Afterwards, the run of the checker took ca 6 minutes, with top RAM usage at ca 700 MB; seven failed assertions were returned, among which three were caused by real bugs.

VII. Future work

The most important short term goal is to modify the decompiler so that CBMC can check more pointer operations. To achieve this, probably some form of type information recovery should be implemented.

Another two features can be implemented with little effort. Firstly, it should be possible to check for integer underflows in the "size" argument to memcpy function, using techniques similar to the above. Secondly, checking for format string bugs should be simple - it is enough to check that the format argument is effectively a string constant (possibly passed through a few levels of function calls).

A more challenging task would be to provide support for C++ code. The main problem is how to handle virtual function calls; it would be interesting to investigate in how many cases it is possible to deduce automatically what the type of an object is, and consequently which virtual function is called at a given place in the code.

VIII. Conclusion

This paper documents the current state of the UQBTng tool. Judging by the performed experiments, it is possible to detect exploitable integer overflow condition, while requiring little interaction from the user. Particularly, it appears that in order to receive good results, it is enough to specify semantics of only a few library function. Further development is needed to increase the tool's capability of handling pointer dereferences of more complex data structures, but even currently the tool can be useful for a security researcher.

References

[1] Cristina Cifuentes and others, *UQBT*, http://www.itee.uq.edu.au/~cristina/uqbt.html

[2] SANS Institute, *The Twenty Most Critical Internet Security Vulnerabilities*, http://www.sans.org/top20/

[3] Bugtraq mailing list, http://www.securityfocus.com/archive/1

[4] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler, *MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties*, Proceedings of the 10th ACM conference on Computer and communication security (ACM CCS), 2003.

[5] Immunity, Inc., *Microsoft Windows: A lower Total Cost of 0wnership*, http://www.immunitysec.com/downloads/tc0.pdf

[6] Remotely exploitable integer overflow in openssh, http://www.openssh.com/txt/preauth.adv

[7] Daniel Kroening, *Bounded Model Checking for ANSI-C*, http://www.cs.cmu.edu/~modelcheck/cbmc/

[8] DataRescue, Inc., *The Interactive Deassembler*, http://www.datarescue.com/idabase/index.htm

[9] Rybagowa, *Seven years and counting*, http://www.lvegas.com/little_cbtc_proceedings/27_02_04.html

[10] Microsoft Security Bulletin MS05-053, http://www.microsoft.com/technet/security/Bulletin/MS05-053.mspx

[11] *Microsoft Portable Executable and Common Object File Format Specifi cation*, www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[12] Microsoft Security Bulletin MS05-046, http://www.microsoft.com/technet/security/Bulletin/MS05-046.mspx