Automatic Heap Exploit Generation

Brent Lim Tze Hao, Advised by Professor David Brumley
Carnegie Mellon University, Pittsburgh, PA
{*brentlim, dbrumley*}*@cmu.edu*

Abstract

The automatic exploit generation (AEG) challenge is, given a program, automatically find vulnerabilities and generate exploits for them. Avgerinos et al showed that, given the source code of the program, AEG was possible for certain stack smashing and format string exploits. In Automatic Heap Exploit Generation (AHEG), we do away with the need for source code and we extend AEG to automatically find heap bugs and generate heap exploits on applications running on Windows XP SP3. Our contributions are: 1) we show how techniques developed by Avgerinos et al generalize to binary-only analysis, 2) we propose memory tagging, a technique used to infer the *class* of data, as opposed to the *type* of data, base on semantic analysis of the program, 3) we introduce "2-steps exploits", which extends Avgerino's approach to exploit generation to heap exploits 4) we build an end-to-end system that takes executables on Windows XP SP3 and automatically generates crashing inputs against them.

1 Introduction

Avgerinos et al showed that it is possible to automatically discover vulnerabilities and generate control flow exploits, given only the source code of the target program [1]. The main idea of his work was to use preconditioned symbolic execution to quickly search for pathological paths in the program and imposing additional constraints on the crashing input so that the solution to these constraints is the exploit string.

This work builds heavily on and extends the work by Avgerinos et al. In particular, this work addresses two of the common criticisms against AEG: 1) AEG was limited to stack-based buffer overflows and format string exploits, which are regarded by some as "trivial" exploits and 2) AEG required access to the source code of the program.

This paper develops techniques and a proof-of-concept for automatic heap exploit generation (AHEG) on executables compiled to run on Windows XP SP 3 on an x86 architecture and stripped of debugging information.

Contributions

1. We develop Simple ASM, a subset of the x86 instruction set rich enough to express real-world programs. We provide the translation from x86 to Simple ASM, and show techniques developed in AEG extend to programs expressed in Simple ASM.
2. We propose *memory tagging*, a technique used to infer the *class* of data, as opposed to the *type* of data, based on semantic analysis of the program. For example, consider *malloc(strlen(get_user_input()))* and *malloc(get_user_input())*. In both examples, the user can control the size of memory allocated by *malloc* and in both cases, the function *malloc* expects an argument of *type* **int**. However, the actual user input to *influence* the same outcome differs. In other words, *the predicate imposed on the user input differs*. For example, to get the program to allocate 5 bytes of memory, in the first case, we might

provide the user input “aaaaa” and in the second case, we might provide the number “5”. Memory tagging is important because in assembly, it is not immediately obvious that the input to a function, in this case *malloc*, comes directly from user input or as a result of operations that are semantically equivalent to *strlen*.

3. We introduce ”2-steps exploits”, an explicit construction of the predicates on the input space to generate exploits against doubly-linked linked list, commonly used in the implementations of heap allocators, including the Windows Heap Manager. Brumley et al showed that exploit generation can be automated by characterizing exploits as predicates on the program state space [2], hence the limitations of AEG to stack-smashing attacks and format string attacks can be overcome by constructing predicates for different classes of exploits.
4. We build an end-to-end system that takes executables running on Windows XP SP3 and automatically generate crashing inputs as proof-of-concept of the techniques introduced in this paper.

2 Background

2.1 Bug Find

In AHEG, we are interested in finding *exploitable bugs*, which are flaws in programs that allow an attacker to perform arbitrary code execution. Formally, we are searching for paths in programs which lead to violations of enforceable security polices [11], in particular, that EIP does not contain user input.

Three popular techniques employed today in software verification to search for such pathological paths are taint analysis [7] [9], symbolic execution [8] [6] [4] [5] [3] and concolic execution [13].

Taint Analysis The idea of taint analysis is to identify certain sources from which user input, also known as *tainted* input, is introduced, such as from sockets, files or command line, and to propagate the *taint* according to a *taint policy*. This technique is a form of data flow analysis and is used to identify the bytes in memory which user has direct control over. A useful application of this technique is to check that EIP is never tainted.

Forward Symbolic Execution Instead of supplying the program with real, or *concrete*, user input, in forward symbolic execution, we emulate the program with *symbolic* bytes. Each time we condition branch on a *symbolic* byte, we fork a new interpreter and explore both branches simultaneously. We also impose contraints on the symbolic bytes at each branch, so that a string satisfying these constraints will take the same path in the program. For more information on taint analysis and forward symbolic execution, we refer the reader to a survey by Schwartz et al [12].

Concolic execution Concolic execution is a variant of symbolic execution, except that instead of emulating the program with *symbolic* user input, we run the program with *concrete* user input. We then instrument the program and treat these *concrete* bytes as *symbolic*, so that whenever we condition branch on user input, we impose an additional constraint, which we can negate and solve for to get another input that traverses a different path in the program. The main advantage concolic execution has over symbolic execution is that in implementing concolic execution, we do not have to keep track of the program state for every branch.

AHEG employs a combination of all three techniques.

2.2 Exploit Generation

In APEG [2] and AEG [1], exploit generation is reduced to the problem of generating predicates on the input space so that the exploit is the satisfying input to the predicates. In AHEG, we take this approach further by explicitly constructing predicates for different classes of exploits. The key insight in constructing such predicates is in encoding the *influence* the user has on specific bytes in memory. The most obvious *influence*

is *direct influence*, in which the byte in memory is exactly what the user entered. Another *influence* is *transformation influence*, in which the byte in memory has been transformed by a series of operations (such as ADD, SUB, etc) from the original user input, so that to control the byte in memory, we have to apply an inverse to the series of operations to get the required user input. This inverse is usually performed by a solver. Both of these *influence* can be identified via data flow analysis and are used in APEG and AEG. However, there exists other *influence* over bytes which user might have indirect control over. To use the same example in the introduction, in *malloc(strlen(user_input))*, the user can control the argument of *malloc* by varying the size of the input buffer. By extending symbolic execution to loops, Saxena et al showed that we can identify such *length influence* [10].

3 Simple ASM

We adopt the usual 32-bit x86 architecture: byte-addressable memory model with 2^{32} entries and a processor with 8 32-bit general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP), an instruction register EIP, and a EFLAGS register. EFLAGS include the Sign Flag (SF), Zero Flag (ZF) and the Overflow Flag (OF). Instructions are stored on memory together with data.

Instructions in Simple ASM are divided into 4 categories:

Data flow instructions These are the binary operations, *MOV*, *ADD*, *SUB*, *MULT*, *DIV*, *OR*, *XOR* and *AND*, which take *destination* and *source* operands (in that order), perform the corresponding operation and save the results in the *destination* operand. An operand is a general purpose register or an immediate value. If the destination operand is an immediate value, the results are not saved. This is to emulate instructions like CMP and TEST, which are essentially SUB and AND instructions respectively, except that the results of the operations are not saved. All data flow instructions might set/unset certain flags in the EFLAGS register, depending on the result of the operation performed, and are governed by the rules provided in the Appendix. Data flow instructions propagate user input, or *taint*, as commonly known in taint analysis.

Control flow instructions These are *UncondJump*, which takes one operand and sets EIP to the value of the operand, and *CondJump*, which takes two operands, the *cond* operand and the *location* operand, checks the *cond* operand against the EFLAGS, and if the conditions are met, sets EIP to the value of the *location* operand. Otherwise, it sets EIP to the address of the next instruction. An example of a *cond* operand is EQ, which is true if the ZF is set. The list of *cond* operands and the corresponding EFLAGS conditions are given in the Appendix. Control flow instructions propagate control flow information, or *implicit flow*.

User input source These are the functions which introduces user input into the system. In SimpleASM, there are only 2 sources of user input, SymFile and SymArgv. *ReadFile* takes 3 arguments, *buf*, *NumBytesToRead*, and *& NumBytesRead*, where *buf* is the memory address of the start of the buffer where user input is copied into, *NumBytesToRead* is the maximum number of user input to copy, and *& NumBytesRead* is the memory address to store the actual number of bytes copied. *SetFilePointer* takes 2 arguments, namely a *destination* operand and one of three values: FILE_BEGIN, FILE_CUR, and FILE_END. The exact behavior of SetFilePointer is detailed in the Appendix, but informally SetFilePointer moves the file pointer of SymFile based on the second argument and saves the position of the file pointer in the *destination* operand. Finally, the last function is *GetCommandLine*() which returns the address of the buffer containing user input. We assume that during the initialization phase, the loader copies user data from SymArgv into a predefined location in memory and *GetCommandLine*() returns the address of that predefined location.

Macros Given an operand x, (x) refers to the value of x. If x is a register, then (x) is the value stored in the register, which is always an immediate value. Otherwise, x is an immediate value and $(x) = x$. Given an immediate value y, $[y]$ refers to the value stored in memory at the address y.

4 Memory Tagging

Definition Let A be a set of user input. Let f be a function from a set of user input A to the set $\mathbb{N} \cup \{\perp\}$. Then the *influence* the user has over a byte i, is a function f_i, such that if $f_i(x) = n, x \in A, n \in \mathbb{N}$, the value of byte i when the program terminates is n. The value of the byte is *undetermined* if $f_i(x) = \perp$. The user has no *influence* over byte i if $f_i(x) = \perp, \forall x \in A$.

Suppose we are given a program P, and we have found a path in the program which performs *Uncond-Jump* (i), where i is some address in memory. Given f_i, we can then ask the question, does there exist $x \in A$, such that $f_i(x) = n$, for some n we choose? If the answer is yes, we have essentially found an input that controls EIP, i.e. a control-flow hijack exploit. We call x the *satisfying input* such that $f_i(x) = n$, and given f_i, we can recover x with the help of constraint solvers. Hence, the goal is to find f_i.

The key idea of *memory tagging* is it encodes the *influence* the user has over a byte and approximates f_i. For each byte in memory, we are interested in the amount of control we have of that byte as a function of user input. In AHEG, we introduce 4 tags, DAT, SYM, LEN and PTR.

4.1 Tag introduction

DAT tag If a byte has been tagged DAT(x, y), where x is a number and $y \in \{SymArgv, SymFile\}$, then that byte is under *direct influence* from the x^{th} byte of user input from the command line or from a file. This means that to set that byte to a particular value, say *42*, we need only set the x^{th} byte of user input to *42*.

Recall that in Simple ASM, there are only two functions that retrieve user input, namely *GetCommand-Line* and *ReadFile*. In both cases, we have a buffer containing user input and we tag each byte in the buffer with *DAT* since these buffers came directly from user input.

SYM tag If a byte has been tagged SYM(y), where y is an expression involving a set A of user input bytes, then the byte is under *transformation influence* from the user input bytes $x_1, ..., x_n \in A$. This means that to set the byte to a particular value, say z, we need to set $x_1, ..., x_n$, so that they evaluate to z in y.

Whenever either operands of a data flow instruction, except MOV, is tagged, we might introduce a SYM tag to the destination operand. The exact rules governing the introduction of SYM tags are listed in the Appendix, but intuitively, the SYM tag "remembers" all the operations performed on user input. For example, given the instruction ADD EAX, EBX, where [eax] = 5, and EBX is tagged with DAT(7, SymArgv), then we will tag EAX with SYM(ADD(5, DAT(7, SymArgv)). If later, we encounter yet another instruction, say SUB ECX, EAX, where ECX is tagged with DAT(5, SymArgv), then we retag ECX with the tag SYM(SUB(DAT(5, SymArgv), SYM(ADD(5, DAT(7, SymArgv)))), which means that ECX really is the difference between the 7^{th} and 5^{th} byte of user input and the sum of the constant 5.

PTR tag The tag PTR(n, p), where n is a unique *buffer id*, and p, an address in memory, tells us that the value of this byte has been used as a pointer to access address p, i.e. this byte has been *dereferenced*. Inferring pointers in the program allows us to perform fat pointer analysis to determine check for unsafe dereferences (such as dereferencing beyond the buffer).

LEN tag To be filled in...

Multiple tags It is possible that a byte can be tagged in more than one way. For example, if a byte that user controls is used to dereference memory, then it will be tagged with DAT and PTR. In these cases, we concatenate the tags DAT . PTR.

4.2 Tag propagation

MOV is the only data flow instruction which guarantees that user input is copied from one location to another unaltered. If the source operand of *MOV* is tagged, then we copy this tag to the destination operand.

4.3 Buffer inference

Some of the analysis we perform, such as fat pointer analysis, requires buffer information, which is available in source code analysis, but not in assembly. Hence, in this section, we describe ways to recover these buffer information, so that we may employ those techniques.

In AHEG, we treat every data structure as a buffer, which we define to be a contiguous sequence of bytes in memory as used by the program. For example, an **int** is a buffer of 4 bytes. Buffer is a semantic (as opposed to syntactic) property of programs;

References

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2011.

[2] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[3] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008.

[4] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, 2005.

[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2006.

[6] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

[7] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[8] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19:386–394, 1976.

[9] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2005.

[10] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution of binary programs. In *International Symposium on Software Testing and Analysis*, 2009.

[11] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[12] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[13] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, 2005.