ISSN 0361-7688, Programming and Computer Software, 2015, Vol. 41, No. 6, pp. 373–380. © Pleiades Publishing, Ltd., 2015.
Original Russian Text © V.A. Padaryan, V.V. Kaushan, A.N. Fedotov, 2015, published in Trudy Instituta Sistemnogo Programmirovaniya, 2014, Vol. 26, No. 3, pp. 127–144

Automated Exploit Generation for Stack Buffer Overflow Vulnerabilities

V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov

Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia

e-mail: vartan@ispras.ru, korpse@ispras.ru, fedotoff@ispras.ru

Received December 15, 2014

Abstract—An automated method for exploit generation is presented. This method allows one to construct exploits for stack buffer overflow vulnerabilities and to prioritize software bugs. The method is based on the dynamic analysis and symbolic execution of programs. It could be applied to program binaries and does not require debug information. The proposed method was used to develop a tool for exploit generation. This tool was used to generate exploits for eight vulnerabilities in Linux and Windows programs, of which three were not fixed at the time this paper was written.

DOI: 10.1134/S0361768815060055

1. INTRODUCTION

As information technologies develop, software security and tools for ensuring the security become more and more important. Complex software is intensively used in critical applications—it controls transport, medical equipment in hospitals, operation of power plants, etc. Failures in the operation of this software can lead to serious consequences, and the intentional malicious use of bugs in software can cause even greater damage. Bugs the use of which can cause the deliberate violation of a system integrity and disturb its operation are called vulnerabilities. Many large IT companies (such as Microsoft, Google, and others) not only support research on bug and vulnerability detection but also practically deploy advanced technologies in the SDLC.

Bugs and vulnerabilities can be detected both at the level of source code and binary code analysis. The latter approach is preferable because abstractions of high-level languages hide specifics of the program operation that are important for detecting bugs and evaluating their severity. In addition, source code is often unavailable. For that reason, computer security experts have to deal with executable (binary) code and use appropriate analysis methods [1]. In recent years, the approach to bug detection based on symbolic execution has been intensively developed.

Symbolic execution was proposed in the end of the 1970s for software testing [2]. The symbolic execution is the execution of a program in which specific values of variables are replaced with symbolic values. Typically, symbolic values correspond to the input data of the program. Operations on symbolic values generate formulas that describe the sequence of operations on symbolic variables and constants. Each conditional branch that depends on symbolic data adds an equation describing the execution flow through a certain branch. The system of equations thus constructed is the path predicate because it describes a scenario of the program execution. This system of equations is passed to a solver in which the symbolic variables are unknowns. The solution of this system of equations is a definite set of values for the symbolic variables.

The idea of symbolic computations was originally aimed at improving testing coverage. However, recently this technique was originally aimed at improving used for guided search of certain program states. Before calling the solver, the path predicate is extended by equations that describe the program state to be achieved. In the context of the present paper, this is the situation in which vulnerabilities are triggered. Due to a large number of vulnerability classes and multiple factors that affect the activation of a vulnerability, attempts to formally describe vulnerabilities at the binary code level were made only for some particular cases.

Usually, vulnerabilities are caused by software bugs. However, not every bug causes a vulnerability. Modern fuzzing tools used in industrial software development produce thousands of inputs that cause abnormal termination [3].

An important issue is bug prioritizing. Bugs that can be exploited should be fixed first. The bugs that allow an attacker to execute an arbitrary code are most dangerous for users and most desired for attackers.

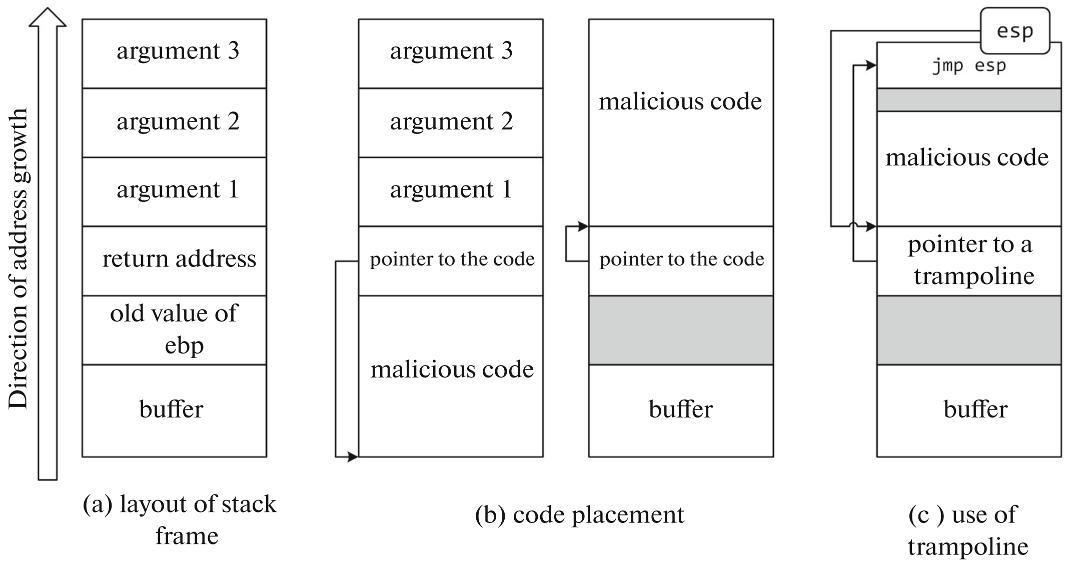


Fig. 1. Stack organization and methods of placing injected code on it.

In this paper, we propose a method for evaluating the detected bugs based on the symbolic execution of binary code. For a given set of input data that bring the examined program to an abnormal termination, an exploit (i.e., a set of input data that exploit the vulnerability) is constructed for a widespread type of vulnerabilities—stack buffer overflow. The bugs for which an exploit could be constructed are classified as critical—they must be fixed as soon as possible. The proposed method can be automated, and we developed a software tool implementing it. This tool allows generating exploits for bugs, so that the shell-code specified by the user is executed.

The paper is organized as follows. The methods underlying the proposed approach are discussed in Section 2. In Section 3, the fundamentals of stack buffer overflow exploits are described. In Section 4, the proposed method is described, and some implementation features are presented in Section 5. In Section 6, the results and directions of future research are discussed.

2. ANALYSIS TECHNIQUES

The binary code can be analyzed using the static and dynamic approach [1]. The symbolic execution within the static approach is limited because of the high complexity of the resulting system of equations. Only the dynamic [4] or combined [5] analysis was reported to be successful.

The studies described in the present paper are based on the capabilities of the binary code analysis environment [6]. The main subject of analysis are traces of machine instructions produced by the full system emulator described in [7, 8]. The traces contain register states and information about interrupts and interaction with peripheral devices, which makes it possible to reconstruct the combined static-dynamic representation of all program images executed in the system and efficiently analyze its properties. The main purpose of the analysis environment is to automate the method of extracting algorithms from binary code [9] and to raise the representation level of these algorithms.

Since the set of input data causing the abnormal termination of the program is known, the execution trace with the abnormal termination can be obtained. To generate an exploit, it suffices to consider only the instructions that deal with the data processing from input moment until the abnormal termination. To select such instructions, a dynamic trace slicing algorithm augmented with the taint analysis is used.

Modern processor architectures contain a lot of various instructions with complex semantics and side effects. A widespread approach that makes it possible to support a variety of architectures is the use of an intermediate representation. We use Pivot intermediate representation [10], which provides a unified description of instruction semantics for various architectures. This intermediate representation satisfies the SSA-form, which considerably simplifies the analysis. The main operators used in Pivot are as follows.

- The operator `NOP` has no any effect.
- The operator `INIT` initializes a local variable by a constant value.
- The operator `APPLY` applies one of the operations. Local variables are used as parameters and the result.
- The operator `BRANCH` transfers control.
- The operator `LOAD` loads a value from an address space.

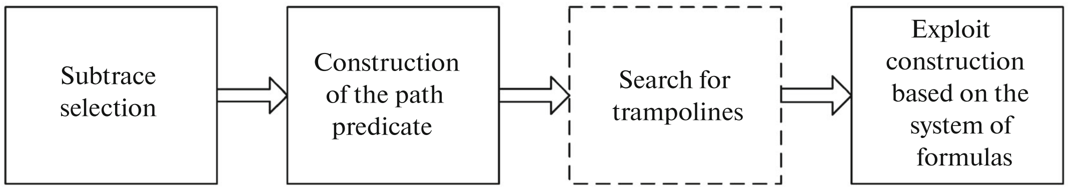

Subtrace selection
Construction of the path predicate
Search for trampolines
Exploit construction based on the system of formulas

Fig. 2. Decomposition of the method into four phases.

• The operator STORE writes a value to an address space.

To describe the memory and registers, Pivot uses the model of unified address spaces. From the viewpoint of this model, all addressable operands of the target CPU architecture (registers, memory, and input-output ports) are placed in linear address spaces. The access to such a space uses a pair (space identifier, offset). To account for side effects, a model status word is used, which is similar to the flag register in the x86 architecture.

3. EXPLOITATION OF STACK BUFFER OVERFLOW VULNERABILITIES

Consider a situation in which the size of data written into a buffer on the stack exceeds the buffer size. Figure 1a shows the stack frame of a function in the situation of buffer overflow. Traditionally for x86, the stack grows from higher addresses to lower ones (from top to bottom in Fig. 1a). The parameters are placed on the stack in reverse order. The call of a function results in placing the return address on the stack after which the function can store the old value of the *ebp* register and then allocate memory for local variables, including a buffer. The data are written to the buffer in increasing order of addresses (from bottom to top in Fig. 1a).

If the size of data written to the buffer exceeds the buffer size, then the memory above the buffer on the stack will be rewritten; in particular, the old ebp value, the return address, and the function arguments can be rewritten. If the attacker can control the values written to the buffer, then he or she can ensure that the return address will contain the pointer to an arbitrary code, which can be formed by the attacker. The execution of such a code can have serious repercussions, including the compromise of the application data, the user, or the operating system. Typically, this code calls the command shell, and it is called shell-code. In reality, the use of shell-code is so popular that any injection payload is called shell-code. The attacker can place this code below the return address or above it (Fig. 1b).

Note that the memory allocated for the stack can be protected from code execution; in this case, an attempt to execute a malicious code will result in abnormal program termination. Then, the attacker can use the return-oriented programming technique [11], which makes it possible to compose shell-code from available code fragments. In addition, address space randomization hampers exploiting the vulnerability by loading the malicious code at different addresses for different attempts. In this case, the value to be written at the place of the return address cannot be predefined. However, often one of the registers points to a stack space at the time of returning from the function. If this space is available for code placing, then the control can be transferred to this code using the instruction jmp ⟨reg⟩ or call ⟨reg⟩ that is located at a known address. The instructions of this type are called trampolines. In this case, upon the return from the function the control will be transferred to the trampoline instruction, and from it to the code placed on the stack (Fig. 1c). Note that the use of trampolines is useful not only in the case of randomization but also when the initial address of the shell-code contains a null byte. Often, buffer overflow occurs when a null-terminated string, which cannot contain nulls, is copied; hence the address cannot be rewritten by the desired value.

4. WORKFLOW

The exploit generation procedure is subdivided into four phases of which one is optional (Fig. 2).

First, a subtrace consisting only of instructions that process the input data is selected. For this purpose, a slicing algorithm and information about the point where the input data are received and the point of abnormal termination are used. For the selected subtrace, the path predicate is constructed. The search for trampolines is the optional phase. Next, the exploit for the shell code specified by the user is generated.

4.1. Subtrace Selection

A subtrace is selected in order to restrict the number machine instructions to be examined. Only the instructions that process the input data directly or indirectly are selected for the subtrace. The subtrace is constructed using the trace slicing algorithm [12]. The slicing algorithm needs the range of trace steps in which the algorithm traces the data and the initial set of traced data. The initial step corresponds to the point where the input data are received, and the final step corresponds to the point of abnormal termination.

Initial step and input data buffer. An analyst can search the input data buffer and the trace step at which the buffer is complete using one of several possible methods.

Many programs get data using library functions. Knowing the data source (e.g., network, file, etc.), the analyst can find calls of the corresponding functions in the trace. For example, to get data from the network, the function `recv` is typically used, and data are read from a file using the function `ReadFile`. The input data can be also passed as command line parameters; in this case, the analyst should find the call of the function `main`.

Search for the program crash step and memory location in which the return address should be rewritten. This task is subdivided into two subtasks:

• search for the memory location where the return address that should be rewritten in the case of buffer overflow is stored;

• search for the trace step at which the control is transferred to the rewritten address.

The memory location is sought using the slicing algorithm; the criterion is the found input data buffer. For each selected instruction, the effective address of the destination operand is compared with the addresses of memory locations that store the return addresses of functions for the current call stack. If these addresses overlap, then the return address of one of the functions in the call stack has been overwritten. In this case, the memory location that stores the rewritten address is the desired memory location.

Next, using the algorithm described below, the steps in the trace at which the abnormal termination seems to have occurred are sought. For each such step, the value of the `ESP` register is compared with the memory address storing the return address (this memory location was obtained at the preceding step). If these values are identical, this step is considered as the step at which the program was crashed.

Information about possible abnormal terminations. The concept of abnormal termination depends on the operating system. Since the method proposed in this paper is designed for working under arbitrary operating systems on arbitrary processor architectures, the search for abnormal termination uses an abstract model of a general-purpose processor. Program crash is recognized as particular event sequence within this abstract model. Note that the abnormal termination almost always occurs as a result of a low level exception in the program. In turn, the exception forces the processor to handle interrupt and transfer control to a proper interrupt handler. The notion of exception is not associated with a particular platform and architecture. Hence, the abnormal termination search algorithm is based on the analysis of exception handling. One must differentiate between interrupts occurring due to instruction exceptions and the interrupts occurring due to the IO operations or other events. Moreover, since the trace reflects the execution of all programs in the system, it contains instructions and exceptions associated not only with the examined application but with all other applications.

The algorithm used in this paper finds the set of trace points at which terminations probably occurred. Since the abnormal termination of the program occurs due to exceptions and exceptions cause interrupts, it is reasonable to search for abnormal termination points among the interrupt trace points. In addition, exceptions violate the natural control flow—an exception handler will return control to process termination routine rather than the next instruction. This observation considerably reduces the number of trace points to be examined.

For each interrupt, addresses of instructions at the entry and exit points are determined. Next, the instruction before the interrupt is considered. Here, two cases are possible—a control transfer instruction and other instructions. For control transfer instructions, the target address must be calculated—the instruction at this address (or at the address after the control transfer instruction if this is a conditional transfer) must be executed next. For all other instructions, the address after the interrupt is compared with the address after the executed instruction. If they are identical, the execution sequence was not broken. Similar checks are performed for control transfer instructions, but the comparison is with the target address.

The result is the set of suspected trace points in which exceptions probably occur. Using the procedure described above, the abnormal termination point is found among these points by direct search.

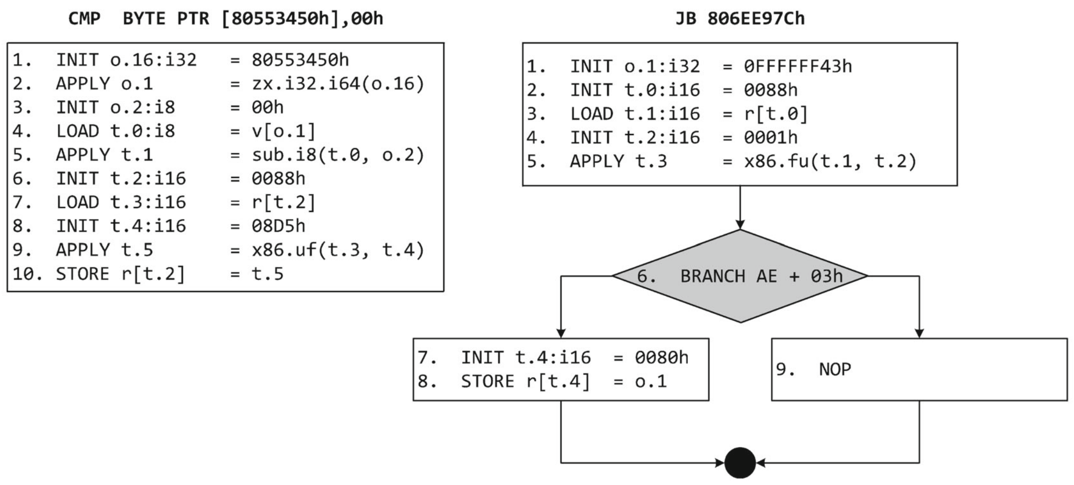
4.2. Construction of the Path Predicate

The path predicate is constructed using trace slicing. The slicing criterion is the input data buffer found above and the trace step where these data are received. The trace analysis is limited by the step where the program is terminated abnormally.

The result is the trace slice and the sets of tracked memory cells for each step of this slice. This data is used to construct the path predicate.

In our case, the path predicate is an SMT-equations set [13] that describes the transformations of the input data at the way to a certain program point. The path predicate is needed to describe constraints on the input data obtained on the path to the termination point.

The path predicate construction consists of two translations. First, a machine instruction is translated into an intermediate representation, and then this representation is translated into the corresponding SMT equations. Memory and registers address spaces is rep-



```
CMP  BYTE PTR [80553450h],00h

1.  INIT o.16:i32   = 80553450h
2.  APPLY o.1       = zx.i32.i64(o.16)
3.  INIT o.2:i8     = 00h
4.  LOAD t.0:i8     = v[o.1]
5.  APPLY t.1       = sub.i8(t.0, o.2)
6.  INIT t.2:i16    = 0088h
7.  LOAD t.3:i16    = r[t.2]
8.  INIT t.4:i16    = 08D5h
9.  APPLY t.5       = x86.uf(t.3, t.4)
10. STORE r[t.2]    = t.5
```

```
JB 806EE97Ch

1.  INIT o.1:i32  = 0FFFFFF43h
2.  INIT t.0:i16  = 0088h
3.  LOAD t.1:i16  = r[t.0]
4.  INIT t.2:i16  = 0001h
5.  APPLY t.3     = x86.fu(t.1, t.2)

6.  BRANCH AE + 03h

7.  INIT t.4:i16  = 0080h
8.  STORE r[t.4]  = o.1

9.  NOP
```

Fig. 3. Translation of CMP and JB x86 instructions.

resented by two arrays of bit vectors. Memory and register arrays in SMT are declared as follows:

(declare-const r_0 (Array (_ BitVec 16) (_ BitVec 8)));

(declare-const v_0 (Array (_ BitVec 64) (_ BitVec 8)));

Here, the address size in the space of registers (r) is 16 bits, and in the space of virtual memory (v) the size is 64 bits. The data granularity in both spaces is 8 bits.

We consider the construction of equations using an example of two consecutive x86 instructions CMP BYTE PTR [80553450h], 00h and JB 806EE97Ch. Figure 3 shows the result of binary translation.

The Pivot instructions are translated one by one, beginning from the first one. The operator INIT is translated into a constant expression in SMT. The operator APPLY is translated into an equivalent operation of the SMT solver. For example, for the second operator APPLY of the instruction CMP, the expression

((_ sign_extend 64) #x80553450)

will be generated.

The operator Load loads the value of an address space location (memory cell or register) into a local variable. We should find out if this element is tracked: the element is sought in the set of tracked elements when the next instruction is selected.

If the set contains the element, then element is assumed to be symbolic; otherwise, its concrete value is used. To obtain concrete values, the buffer reconstruction algorithm is used. There are situations when the value could not be reconstructed. Then, the element is considered as a symbolic one. Let the memory byte at the address 0x80553450 be symbolic; then, the following expression will be generated for the operator Load:

(select v_0 ((_ sign_extend 64)#x80553450))

As a result, for the fifth APPLY operator, we obtain

(bvsub (select v_0 ((_ sign_extend 64) #x80553450)) #x00)

The instructions from the sixth to the tenth ones are not processed because they update the flag register. To avoid redundancy in the equations, we use lazy flag calculation. The results of translation of the machine instruction CMP is the expression for the fifth operator and internal data needed to set flags. Now, consider the construction of equations for the instruction JB 806EE97Ch. The operators from the second to the fifth ones are not processed because the flag register r:[0x88] is not symbolic. The operators 1, 7, 8, and 9 are not processed because the instruction counter r:[0x80] is not symbolic either. Thus, only the sixth instruction Branch is processed.

For the condition of Branch, the required flags are calculated, and they are loaded into the model status word. In this case, the condition is AE (greater than or equal to for unsigned numbers). The flag CF is calculated and loaded into the model status word. Next, an equation for the execution of the instruction Branch is formed. In the case under consideration, the flag CF must be unset. Then, an equation reflecting the execution or nonexecution of conditional control transfer is added. Let the conditional branch was executed in the trace. Then, to execute the jump instruction JB, the equation (CF=0) = false must be added, which indicates that the Pivot operator Branch AE is not executed. The constructed equation becomes a part of the path predicate.

4.3. Search for Trampolines

To find trampolines, an analyst must specify which modules were loaded into virtual memory of analyzed applications. The base load addresses for these modules are known a priori. In the codes of these modules, the trampoline instructions will be sought. These instructions are `jmp Reg` or `call Reg`. They transfer control to the address specified in the register operand. First, it is reasonable to determine memory buffers in which shell code can be potentially placed. From the set of tracked (tainted) memory locations at the time of program crash, the memory locations that form continuous buffers of the size suitable for shell-code placement are selected. If there are such memory locations, registers that point into these locations are selected. If there are no such registers, the trampoline search is terminated. In the case of success, trampoline instructions are sought for the selected registers. The binary code (opcode and operand) of such instructions occupy two bytes that should be found in any executable sections of the modules. Note that the needed bytes can even be located on the boundary of two different instructions. As the size of the sections looked through increases, the probability to find at least one trampoline rapidly increases. For 500 Kb section, this probability is as high as 0.999.

4.4. Exploit Description and Solutions of the System of Equations

To generate an exploit, the path predicate should be extended with the equations that describe this exploit. Equations for control hijack after stack buffer overflow can be divided into two types:

• equations for placing the shell-code in a memory buffer controlled by the attacker;

• control transfer to a memory buffer controlled by the attacker.

For the memory buffers that are tracked at the point the program crash occurs, we choose the buffers that are larger than or equal to the size of shell-code. If there are no such buffers, then we conclude that this vulnerability cannot be exploited with such a payload. Otherwise, we build up the following equation for one buffer. Let the shell-code consist of a string “`ABCD`” and the address range be from `1000` through `1003`, respectively. The equation is

```
v(1000,1) = 0x41 ∧ v(1001,1)=0x42 ∧
v(1002,1)=0x43  ∧  v(1003,1)  =0x44,
where ∧ is the conjunction.
```

To describe the control transfer to the memory buffer controlled by the attacker, we should build up an equation to describe that fact that the memory location storing the return address from the function should store the shell-code address.

Let `x` be the address at which the return address is stored and let `l` be the shell-code or trampoline address. Then, the formula has the form

```
v(x,1) = |[0] ∧ v(x+1,1) = | [1]} ∧
v(x+2,1) = | [2] ∧ v(x+3,1)= | [3],
where ∧ is the junction.
```

By combining the placement and control transfer equations with the path predicate, we obtain a set of equations that are sufficient for generating a working exploit. Next, these equations are passed to the SMT-solver, and if the system of equations is consistent, its solution is a working exploit.

Table 1. The list of analyzed applications

Operating system;	Application;	Vulnerability.
Linux	corehttp 0.5.4	CVE:2007-4060
MCBC	libpng (konqueror)	CVE:2004-0597
Windows XP SP3	SuperPlayer 3500	EDB-ID:27041
Linux	iwcongfig v26	CVE:2003-0947
Windows XP SP2	lhhtpd 0.1	CVE:2002-1549
Linux	getdriver (sysfsutils)	
Linux	mkfs.jfs (jfsutils)	
Linux	alsa_in (jack)	

Table 2. Results of the exploit generation algorithm

Operating system	Application	Slice size	Data size, byte	Solution time, s	Total execution time, s
Linux	corehttp 0.5.4	18293	565	1024	1367
MCBC	libpng (konqueror)	2493	536	8	128
Windows XP SP3	SuperPlayer 3500	4855	594	<1	66
Linux	iwcongfig v26	124	80	<1	7
Windows XP SP2	lhhtpd 0.1	20174	320	18	245
Linux	getdriver	152	272	2	41
Linux	mkfs.jfs	209	407	3	23
Linux	alsa_in	241	58	<1	40

5. IMPLEMENTATION DETAILS AND EVALUATION

The proposed method was implemented as a plug-in for the binary code analysis environment. It uses such capabilities of the environment as raising the representation level, general-purpose processor model, and trace slicing. A third-party SMT solver integrated into the plug-in is used to solve the system of equations. Presently, a lot of solvers are available, such as MiniSat, OpenSMT, STP, Yices, Z3, and others. We used the Z3 due to the following advantages:

• incremental approach to the solution of equations;

• support of many data types, including machine-level data types;

• there is a C API that allows one to directly invoke the equation solver, which is much more efficient than the work with text input;

• the source code under the MSR-LA license is available;

• it is faster than other solvers.

We evaluated the developed tool on a number of examples. The 32-bit operating systems Windows XP SP2, Windows XP SP3, Arch Linux (as of April 2014) and Mobile System of the Armed Forces 3.0 (MSAF) were used as guest OSs. Applications with known vulnerabilities were used, as well as applications from the Arch Linux distribution in which bugs were found with the help of black-box fuzzing. The list of analyzed applications is presented in Table 1.

Fuzzing is based on the method described in [14]. The fuzzer starts the analyzed application with all possible single-letter command line parameters (from `–a` to `–z` and from `–A` to `–Z`) and an additional parameter 6676 bytes long. For 6607 applications, the fuzzer obtained 748 crashes for 42 different applications. Among these 42 applications, we selected three applications distributed in three popular software packages `sysfsutils`, `jfsutils`, and `jack`.

Table 2 presents the results produced by the exploit generation algorithm—the size of slice used to process the input data, the size of input data buffer, the time taken by the generation of the system of equations, and the time taken by its solution.

Note that in order to exploit the stack buffer overflow vulnerability using the modern Linux distribution, some protection mechanisms were switched off.

• Address space randomization was switched off.

• The applications were compiled with the flags - without line break and `–U_FORTIFY_SOURCE`; as a result the canary stack protection mechanism and the use of safe functions available in the `gcc` compiler were switched off.

• For the analyzed applications, code execution on the stack was turned on using the `execstack` utility.

In other operating systems (Windows XP and MSAF) we didn't turn off any protection mechanisms. To generate exploits for the Windows XP applications listed in Table 1, trampolines were used because the stack memory addresses contained a zero byte; therefore, the shell code address could not be written directly to the return address. For Linux applications, this difficulty does not arise, and the use of trampolines to overcome randomization was unsuccessful because the only nonrandomized code fragments belonged to the applications themselves, due to the tiny size of applications, the trampoline search algorithm had not find anything. The list of results does not contain the applications for which no working exploit was generated. In these applications, the stack frame included other variables in addition to the overflowed buffer; and the modification of these variables resulted in premature termination, thus no injected code was executed. As a rule, such variables contained pointers, and after they have been rewritten, dereferencing of pointers resulted in access violation. In other cases, a loop control variable was rewritten, which resulted in reading data from an incorrect address. To make the exploit operate under these conditions, the corresponding memory locations must be rewritten with correct values.

6. CONCLUSIONS

An exploit generation method for detected bugs is presented. The method is based on the symbolic execution of binary code; it overcomes address space randomization using trampolines and automate some manipulations that cannot be performed without the user. The method is implemented as a plug-in of the binary code analysis environment. Its application helps the developer to select security critical bugs that must be corrected first of all.

Close results were obtained at Carnegie Mellon University. In [14], the first system for the automatic exploit generation (AEG) was presented. Potentially exploitable vulnerabilities are sought in the source code, and the exploit is constructed at the binary code level. MAYHEM [4] significantly advances the results of AEG. It is designed to find vulnerabilities and uses only stripped binary code. These systems iterate over different execution paths using symbolic execution and detect exploitable program crashes. Unfortunately, all the tools developed by this research group are not available. We also note that these tools also assume that some code protection mechanisms are switched off.

There are also other symbolic execution systems working with the binary code—BitFuzz [15], FuzzBall [16], S2E [17], SAGE [18], Avalanche [19], and others. Most of them are primarily designed for searching execution paths, but they cannot generate exploits.

The results presented in this paper form a complete method that helps prioritize the detected bugs. The generation of an exploit reliably classifies the bug as a critical one. In future, we plan to extend our tool by ROP compilation, which allows one to overcome the non-executable stack protection, exploit other types of vulnerabilities, and implement the method for other processor architectures.

REFERENCES

1. Tikhonov, A.Yu. and Avetisyan, A.I., A combined (static and dynamic) analysis of binary code, *Tr. Inst. Sist. Program. Ross. Akad. Nauk*, 2012, vol. 22, pp. 131–152.
2. King, J.C., Symbolic execution and program testing, *Commun. ACM*, 1976, no. 19, pp. 385–394.
3. Miller, C., Caballero, J., Johnson, N.M., Kang, M.G., McCamant, S., Poosankam, P., and Song, D., *Crash Analysis with BitBlaze*, BlackHat, 2010.
4. Cha, S.K., Avgerinos, T., Rebert, A., and Brumley, D., Unleashing MAYHEM on binary code, in *IEEE Symposium on Security and Privacy*, 2012.
5. Avgerinos, T., Rebert, A., Cha, S.K., and Brumley, D., Enhancing symbolic execution with veritesting, in *36th Int. Conf. on Software Engineering*, 2014, pp. 1083–1094.
6. Padaryan, V.A., Getman, A.I., Solovyev, M.A., Bakulin, M.A., Borzilov, A.I., Kaushan, V.V., Ledovskikh, I.N., Markin, Yu.V., and Panasenko, S.S., Methods and software tools supporting a combined analysis of binary code, *Tr. Inst. Sist. Program. Ross. Akad. Nauk*, 2014, vol. 26, no. 1, pp. 251–276.
7. Dovgalyuk, P.M., Fursova, N.I., and Dmitriev, D.S., Prospects of using the deterministic replay of the virtual machine operation for ensuring computer security, in *Materialy konferentsii RusKripto'2013* (Proc. of the Conf. RusCrpto'2013), Moscow, 2013.
8. Dovgalyuk, P.M., Makarov, V.A., Padaryan, V.A., Romaneev, M.S., and Fursova N.I., Application of software emulators for the analysis of binary codes, *Tr. Inst. Sist. Program. Ross. Akad. Nauk*, 2014, vol. 26, no. 1, pp. 277–296.
9. Tikhonov, A.Yu., Avetisyan, A.I., and Padaryan, V.A., A technique for extracting the algorithm from binary code based on dynamic analysis, *Inf. Security Probl. Comput. Syst.*, 2008, no. 3, pp. 66–71.
10. Padaryan, V.A., Solovyev, M.A., and Kononov, A.I., Simulation of operational semantics of machine instructions, *Program. Comput. Software*, 2011, vol. 37, no. 3, pp. 161–170.
11. Schwartz, E.J., Avgerinos, T., and Brumley, D., Q: Exploit hardening made easy, in *Proc. of the USENIX Security Symposium*, 2011.
12. Tikhonov, A.Yu. and Padaryan, V.A., Application of program slicing for the analysis of binary code represented by execution traces, *Materialy XVIII Obshcherossiisckoi nauchno-tekhnicheskoi konferentsii "Metody n tekhnicheskie sredstva obespecheniya bezopasnosti informatsii"* (Proc. of the All-Russia Conf. on Methods and Tools for Data Security), 2009, p. 131.
13. Ranise, S. and Tinelli, C., The SMT-LIB format: An initial proposal, *Proc. of PDPAR'03*, 2003.
14. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., and Brumley, D., AEG: Automatic exploit generation, *Commun. ACM*, 2014, vol. 57, no. 2, pp. 74–84.
15. Caballero, J., Poosankam, P., McCamant, S., Babic, D., and Song, D., Input generation via decomposition and re-stitching: Finding bugs in malware, in *Proc. of the ACM Conf. on Computer and Communications Security*, Chicago, 2010.
16. Martignoni, L., McCamant, S., Poosankam, P., Song, D., and Maniatis, P., Path-exploration lifting: Hi-fi tests for lo-fi emulators, in *Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, London, 2012.
17. Chipounov, V., Kuznetsov, V., and Candea, G., S2E: A platform for in-vivo multi-path analysis of software systems, in *Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.
18. Godefroid, P., Levin, M., and Molnar, D., Automated whitebox fuzz testing, in *Proc. of the Network and Distributed System Security Symposium*, 2008.
19. Isaev, I.K., Sidorov, D.V., Gerasimov, A.Yu., and Ermakov, M.K. Avalanche: Application of dynamic analysis for the automatic detection of bugs in programs that use network sockets, *Tr. Inst. Sist. Program. Ross. Akad. Nauk*, 2011, vol. 21, pp. 55–70.

Translated by A. Klimontovich