Autonomous Exploitation of System Binaries using Symbolic Analysis

Joran Honig
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.j.honig@student.utwente.nl

ABSTRACT

At the moment many software systems have bugs, and developers can be overwhelmed with bug reports describing crashes or unexpected behavior. Finding the critical bugs to focus on can be time consuming, and might lead to security critical bugs remaining unresolved for an extended period, which in turn can lead to data leaks or improper functioning of important systems.
Current state of the art autonomous methods are still unable to find all bugs, and are often unable to determine if they are security critical. Therefore it is important that methods are developed and improvements are made with automatically finding and validating security vulnerabilities.
In this paper, the application of concolic analysis and constraint solving are applied to this problem. Three algorithms, used to determine exploitable constraints, will be proposed and evaluated. Furthermore, these algorithms will be compared to the current state of the art, providing an overview of the field.

Keywords

Proof Of Concept, Automatic, Exploit, Concolic Testing, Symbolic analysis

1. INTRODUCTION

Many systems have the problem that they have bugs, resulting in undefined and possibly unwanted behavior. Some of these bugs might even turn out to be exploitable. This means that it is possible to craft an input for the software, or a series of inputs, that cause the program to execute code provided by the attacker, or allows the attacker to control key features in a system. An example use of an exploit could be, to install a virus on the machine that is currently hosting the software. Moreover, an attacker can use an exploit to cause an application to grant him or her higher system privileges.

It is therefore important that bugs, and more specifically security critical bugs are discovered and patched, ensuring that systems remain uncompromized. Moreover, automating this process would be even more advantageous, since it would allow for more frequent and thorough analysis of software.

27^{th} *Twente Student Conference on IT* July 7^{st}, 2017, Enschede, The Netherlands.

Nowadays the development of exploits for binaries requires a human, knowledgeable in modern exploitation techniques. There are methods to automatically generate exploits [4, 2, 6]. Although the Cyber Grand Challenge organized by DARPA led to several published papers [12, 4], most exploit generation techniques have only been applied to stack based vulnerabilities[7] and are therefore unable to validate all vulnerabilities. In this paper, a method will be developed which will attempt to generate exploits for software based on existing analysis tools described in section 3.1.

The process of exploit development can be divided in roughly three steps. Namely:

1. Bug or vulnerability discovery
   Where researchers try to find bugs in existing code.
2. Rudimentary Proof of Concept exploit creation, hijacking control-flow
   In this part of exploit development researchers construct an example value that, when provided as an input to the program, takes over the execution of the program.
3. Exploit hardening against exploit mitigation techniques
   There are some defenses against exploitation, to ensure that an exploit works it will need to be adjusted accordingly. Exploit mitigation techniques will be discussed in section 3.3.

Some steps of the process can be sped up with the use of tools. For example, fuzzers and dynamic analysis can be used to speed up the process of finding bugs. There are also tools that help researchers with hardening exploits, like rop chain [8] generators, which are discussed in section 3.2.2, or methods proposed in Q[10]. These methods still require researchers to manually validate exploits and compose large parts of an exploit themselves, slowing down the process of validating vulnerabilities.

Although progress has been made on automating the entire process of exploit development, research has been focused on finding vulnerabilities instead of specifically creating exploits [4, 2, 12].

In this research we will focus on developing a method that is able to automatically generate an exploit, based on a crashing input. This research can be applied and used to speed up and automate the second step of the described exploit development steps.

2. RESEARCH QUESTION

The research question of this research is:
To what extend can concolic execution be used to create

exploits for binaries?
The sub questions for this research are:

1. How can constraints determined by concolic execution be used to generate exploits for heap overflows?

2. How does using symbolic execution and constraint solving to generate exploits compare to existing methods proposed in [2]?

3. BACKGROUND

In this section the relevant topics from exploit development and formal analysis methods will be explored to provide a basis for the other sections. In subsection 3.1 the topics conserning software analysis will be explored. While in the other two subsections will explore attack techniques, subsection 3.2, and exploit hardening methods, subsection 3.3.

3.1 Analysis Methods

Several methods have been developed to test and analyze programs, some of these methods provide an excellent insight in the execution state and logic of a program and can be applied to vulnerability research and development. In the following section we will discuss some of these methods.

Algorithm 1

```
1: function MAIN(X)                          ▷ Where X is a string
2:     if X[0..4]! = "abcd" then
3:         return false
4:     Let A[0..20] be a new array
5:     for i = 0 to len(X) do
6:         A[i] = X[i]
7:     return true
```

3.1.1 Concrete Execution

Concrete execution or analysis is the execution of a program using concrete values. Take algorithm 1, in this case all strings are possible inputs to the program. It is possible to determine the program context and memory state at any moment during the execution. For example, using a debugger an program could be executed and paused on individual instructions, allowing for analysis of the program state.

3.1.2 Taint analysis

Another form of dynamic analysis is taint analysis. Using taint analysis means that some inputs are marked as tainted. While stepping though a program, we can look at which computations are affected by the tainted values[9]. Using this information, it is possible to determine which parts of the input cause the current control flow. Take program 1, taint analysis methods are able to determine that character zero up to four of the input are used to determine which path to take. Note that this method does not provide enough insight to determine the value required to take another path in the program, because it is unable to reason why and how a change in the input causes a different path to be taken.

3.1.3 Symbolic Execution

Instead of executing programs in the concrete domain, during symbolic execution programs are executed in the symbolic domain. This means that instead of supplying the program with concrete existing values we give it symbols. Moving through the program we adjust the expressions for memory locations accordingly.

Take for example array A at line 7 in algorithm 1. The expression for this variable can be described by A = X if the first letters of X are "abcd", otherwise A would be non existent. In this case X is a symbol that represents the user input.

If the execution engine encounters a branch, for example an if statement, it follows each path and maintains the constraints and expressions for each. This type of analysis suffers from some problems. The first problem is that it does not execute as fast as concrete execution. Secondly, it suffers from the phenomena called state explosion, which happens if there exists a program structure with many or possibly infinite states. In this case it is impossible to symbolically execute the entire program. Mitigations to this problem have been proposed in [4, 2, 12, 6] among others. While these mitigations might delay the problem of state explosion, they are not yet able to mitigate it.

3.1.4 Concolic testing

Concolic analyisis is a combination of symbolic execution and concrete execution. While executing a program with a concrete input, the constraints that make the program follow the path that it takes, will also be collected. Take algorithm 1, if we use input "hello", the execution path follows lines one, two and three and then returns false. At line three the following constraint is collected, the input X can not start with "abcd". The collected constraints can then be used to determine other inputs that follow the same path or other inputs that do not.

This method has two advantages. The first being that it is faster or at least as fast as analyzing an entire program, while still having a thorough understanding of why the program went through certain states. In addition to being faster than symbolic execution, this method also does not suffer from state explosion. The downside of this method is that the understanding of the program is limited compared to symbolic execution, since we only follow one path.

3.1.5 Fuzzing

Fuzzing is a concrete analysis method that is used to find bugs in software. A fuzzer works by creating inputs for a program and executing the program with this input. Simple fuzzers use random values to generate new inputs, while more sophisticated fuzzers use methods like taint analysis to reason about new inputs that might cause the program to take alternate paths. The fuzzer then checks if the program has crashed and tries another input, storing all crashing inputs. These crashing inputs can then be used by developers or researchers to determine the cause of the crash. In addition to that, other analysis methods can be used to further explore the cause of the crash.

3.2 Attack Techniques

There are several techniques that can be used while building an exploit, in this subsection some of these exploitation techniques will be discussed. The first subsection describes shellcode, which is the technique used by the algorithms described in section 5. The other subsections show relevant alternatives for this exploitation method.

3.2.1 Shellcode

Shellcode is a set of instructions often with the purpose of taking over the functionality of a program. Shellcode is often the target of the instruction pointer after the control flow of an application has been hijacked.

3.2.2 Return Oriented Programming

In this case the attacker uses instructions already in the code, he does this by changing the instruction pointer to the address at the start of a chain of instructions. These are executed by the program until a return instruction is executed. The attacker will have set the stack to then contain an address of the next instruction chain that he wants executed, repeating this process. This allows an attacker to execute his code. This attack is more thoroughly discussed by Roemer et al.[8].

3.2.3 Ret2libc

This attack is similar to the previous one in that it reuses code readily available instead of running its own. Using this attack, an attacker fills the stack with the arguments necessary for a call in libc and then overwrites the instruction pointer to point to a method in libc, which would then get executed using the provided arguments.

3.3 Exploit hardening

To prevent attackers from successfully exploiting a binary several exploit mitigation techniques have been developed. Although they are effective at mitigating non sophisticated attacks, they are imperfect and there exist techniques to circumnavigate the mitigation techniques.

3.3.1 W xor X

This exploit mitigation technique has been implemented to ensure that arbitrary code, that has been injected into the memory can not be executed. It does so by ensuring that memory is not writable and executable at the same time.

However by reusing existing instructions in the target binary, also called rop gadgets, an attacker could change the status of memory from writable to executable. In addition to that, the existing rop gadgets might also provide the attacker with enough versatility to create an exploit without using shellcode.

3.3.2 Address Space Layout Randomization

To prevent the use of the previous circumvention, the address space layout is randomized in programs with ASLR. This ensures that the attacker is not able to determine what the address of certain key parts of memory will be [11]. However, this technique has also been compromised. In the case of 32 bit binaries the entropy of the random address locations is too low and an attacker is able to brute force it. In addition to that, the "information disclosure" class of vulnerabilities allows attackers to gain information about the layout of the memory, thus negating this mitigation technique.

4. RELATED WORKS

There have been several papers published in the field of automatic exploit generation [12, 4, 2, 6]. Most of these researches develop methods that can be used to automate the entire process of vulnerability detection and exploit generation, focusing on the development of an efficient, scalable method to find bugs and exploit them. And, although exploit generation is still an important aspect, thorough research into the creation of exploits has not been done.

The CRAX paper [6] proposes a method that is very similar to the method in this paper, as they also look into generating exploits based on concolic analysis of crashing inputs. The difference is that we will provide thorough description of the algorithms used to generate exploits and look at modern heap exploitation techniques.

AEG[2] and Mechanical Shellphish[12] have focused on optimizing the bug discovery process, and have therefore not thoroughly researched exploit generation algorithms. They use a method that uses the stack and memory layout information when a crash occurs, to create an exploit. The algorithm used by AEG and Mechanical Shellphish can be found in the AEG[2].

As pointed out in CRAX[6], this algorithm is flawed and fails to generate working exploits in certain situations. It fails to generate a working exploit because it is not able to reason about the instructions that happen after a buffer overflow has occurred[6]. The methods in this paper and the ones described in CRAX are able to correctly generate an exploit in these situations, because it has a complete understanding of all mutations on the memory until the crash happens.

5. ALGORITHMS

We looked at three target vulnerability categories, namely pointer overwrites, return address overwrites and vulnerabilities with an arbitrary write. As both stack and heap overflows can fall in to each of those categories and the algorithm for exploiting these categories might differ, these categories are used instead of just distinguishing between heap and buffer overflows.

In the following sections the internals of these vulnerability classes will be discussed in addition to the algorithm used to generate exploit constraints. The exploits are built up by first finding a location in memory to store the shellcode, subsection 5.1 describes the algorithm to do this. The algorithms for specific vulnerabilities will then be used to determine the constraints necessary to redirect execution to the shellcode.

5.1 Shellcode

To make the program run shellcode, it needs to be placed in memory. Algorithm 2 does this by returning a list of constraints that constrain the memory to contain the shellcode.

Algorithm 2

Input: Π_{bug}: Constraints found using concolic execution
Input: S: Crashing state, contains expressions for memory and registers
Input: ST: Shellcode string that the exploit should execute
Output: SC: a list of tuples where the first element is a set of constraints that ensure the shellcode is in memory in state S and the second element is the middle of the constrained nopsled

1: **function** ShellcodeConstraints(Π_{bug}, S, ST)
2: $SC = []$
3: **for** MA in $S \rightarrow memory$ **do**
4: len = length of symbolic writes from MA
5: **for** $i = 0$ to len - $ST.length$ **do**
6: $\Pi_{shellcode}$ = memory at MA is $i * nopinstruction + ST$
7: **if** $\Pi_{shellcode} \wedge \Pi_{bug}$ is satisfiable **then**
8: $SC+ = (\Pi_{shellcode}, MA + i/2)$
9: **return** SC

5.2 Exploit reliability

Because the analyzed memory addresses may not match the addresses found during the analysis, exploits may not always work. Therefore it is important that the reliability of the exploits is increased. A method to increase the

reliability of an exploit is an exception handler overwrite, where an attacker overwrites the address of an exception handler. Another is a jump to register exploit, where the ip is constrained to an instruction that jumps to the address in a register, and finally the use of a nopsled can increase the reliability of an exploit. The last method has been implemented in the tool and algorithm 2 used to validate this research. Using a nopsled means that the shellcode is prepended with a series of valid instructions that do nothing but increment the program counter. Instead of trying to jump to the start address of the shellcode, the exploitation algorithms will constrain the address to be in the middle of the nopsled, which allows for some inaccuracy of the analyzed memory addresses. Line 6 in algorithm 2 adds the nop instructions to the shellcode.

5.3 Pointer overwrites

In some programs, pointers to methods or functions are stored in memory, these pointers can then later be used to execute the function that they reference. Some vulnerabilities allow an attacker to overwrite a function pointer. Later when the program calls the function pointer, it will execute the code at the new address. An attacker can craft his input in such a way that the function pointer points to a section of memory where he has placed shellcode, thus exploiting the target binary. The algorithm explaining the steps to exploit such a vulnerability is algorithm 3.

Algorithm 3

Input: Π_{bug}: Constraints found using concolic execution
Input: S: Crashing state, contains expressions for memory and registers
Input: ST: Shellcode string that the exploit should execute
Output: Π_{expl}: A set of constraints that, if satisfied exploit the analyzed program

```
1: function BuildExploit(Π_bug, S, ST)
2:     SC = ShellcodeConstraints(Π_bug, S, ST)
3:     for (Π_shellcode, MA) in SC do
4:         # Π(S → ip == MA) is the constraint which
5:         # constrains the program counter to the shellcode address
6:         Π_expl = Π(S → ip == MA) ∧ Π_shellcode
7:         if Π_expl satisfiable then
8:             return Π_expl
```

5.4 Return address overwrites

When a method is executed some variables are put on the stack, one of these variables is the return address of the method. The return address is the address of the instruction that the program should return to after it has finished executing the method. Some vulnerabilities, possibly resulting from out of bounds memory writes, allow an attacker to overwrite this return address. The same method as discussed in the previous section can then be used to exploit the binary. The attacker can change the return address to the address of shellcode and in that way exploit the program. This is similar to the previously discussed method, therefore algorithm 3 is also applicable for this vulnerability class.

5.5 Arbitrary writes

Some programs allow for a, possibly unintended, symbolic write, which means that an attacker can overwrite an arbitrary piece of memory. This can be used to leak information from the target program and also to take over the execution. The latter option will be discussed in this section. To exploit this kind of vulnerability, we constrain the target address of a symbolic write to a function pointer or return address. After the symbolic write, the same approach used for the previous vulnerability classes can be used.

One way that an arbitrary write can be exploited is an overwrite of the General Offset Table, GOT for short. The GOT is used by the program to find the locations of methods in libraries. Every time a library method is called the program finds the address of this method by looking in the GOT. By overwriting an entry in the GOT an attacker can redirect the execution flow of the program to shellcode. After having constrained the target address to a GOT entry, the approach will be the same as for algorithm 3.

It is also possible to overwrite values on the stack. However, since the addresses on the stack are unpredictable only GOT entry overwrites will be discussed in this paper.

Algorithm 4

Input: Π_{bug}: Constraints found using concolic execution
Input: S: Crashing state, contains expressions for memory and registers
Input: ST: Shellcode string that the exploit should execute
Output: Π_{expl}: A set of constraints that, if satisfied exploit the analyzed program

```
1: function HeapExploit(Π_bug, S, ST)
2:     CS = []
3:     if S → symbolic_writes.length > 0 then
4:         for SW in S → symbolic_writes do
5:             for each methodCall after SW do
6:                 CS += (Π(WriteAddress == MethodAddress), methodCallState)
7:     for C, MS in CS do
8:         if C ∧ Π_bug is satisfiable then
9:             Π_expl = BuildExploit(MS, C ∧ Π_bug, ST)
10:            if Π_expl is not Null then
11:                return Π_expl
```

6. VALIDATION

In this section the performance of the algorithms will be discussed. In order to use these algorithms on vulnerable binaries an implementation was made using the framework *angr* [1]. The development process will be briefly discussed in section 6.1. In section 6.2 the experiments ran using the tool will be discussed. Finally arbitrary write vulnerabilities will be discussed in section 6.3.

6.1 Implementation

The implementation of the algorithms in section 5 has been built using the symbolic analysis framework *angr*[1]. The developers of this library have also implemented autonomous exploit generating software[12]. Some of the code which organizes the data found during symbolic execution has been used in our implementation to validate the algorithms used in this research.

The first part of the implementation is a module which simulates a concolic trace on a binary using a crashing input. It does so by constraining the input variables, namely the command line arguments and the standard input datastream, to the given crashing input. It then steps through the program, ignoring any path with unsatisfiable constraints, and often following only one path.

The constraints and memory expressions found using the analysis module are used by the tool to determine if the path taken is exploitable and it finds the constraints for an exploit using the algorithms in section 5.

Angr is at this moment unable to symbolically execute the methods that manage the memory on the heap in libc malloc, it is therefore unfortunately impossible to properly analyze these methods, using this framework. However, in section 6.3 we will show that exploitation using these methods falls in the categories described in section 5, and therefore should be exploitable by the algorithms described in the same section. This means that the implementation is not yet able to generate exploits for vulnerabilities caused by methods managing the heap. Once *angr* supports analysis of these methods, our implementation should work on these vulnerabilities.

Another problem encountered during the implementation was a bug in *angr*, which meant that it was impossible to record method calls during analysis. This is required for the algorithm described in 5.1 to determine the target of a symbolic write. Unfortunately the tool is therefore unable to exploit this vulnerability class.

6.2 Experiments

In order to validate the correctness of the algorithms, we used the implementation to analyze and generate exploits for some example simple vulnerable programs. For the first vulnerability class defined in section 5 a stack and heap based vulnerable program was constructed and tested. Two programs, each with a stack based return address overwrite, have been constructed and tested using this methodology. One where the overflowed array was reversed before the return statement and one where it was not. Finally, the arbitrary write vulnerabilities will be discussed in section 6.3.

All the vulnerable programs are built with the same structure which is displayed as algorithm 5.

Algorithm 5

```
1: function VULNERABLE(INPUT)
2:     Allocate memory which will contain user input
3:     Setup requirements for a vulnerability
4:     Vulnerable statement
5:     Possible memory mutations
6:     Trigger vulnerability
```

The vulnerable programs were then analyzed using the GNU debugger [5] and the exploit generated by the tool. In most cases the tool generated an exploit that would correctly generate an input which would allocate the shellcode to memory and it would constrain the program counter to move to the address of the shellcode. In some cases the expected address of the shellcode was too inaccurate and the exploit would not work. Because of the reasons discussed in 6.1 we can not correctly perform analysis on programs that use the heap, therefore the exploit generated for a pointer overwrite on the heap did not work correctly.

Although the exploit did not manage to cause every program to execute arbitrary code, it did allocate shellcode and managed to constrain the program counter to the estimated address of the shellcode. The only problem being the inaccuracy of the memory addresses. Methods to improve the reliability of exploits have been discussed in section 5.2.

6.2.1 *Pointer overwrites*

The experiment concerning a pointer overwrite was unsuccessful for the vulnerable heap program, since *angr* did not account for the chunk metadata and therefore had mistaken the location of the function pointer. This minor difference was not caused by the algorithm in section 5.3. This should behave correctly once the angr library is able to correctly symbolically execute the methods in malloc.

In the case of a stack based pointer overwrite, the implementation was able to generate an exploit for the vulnerable program.

6.2.2 *Return address*

The algorithm in 5.3 was able to generate an exploit for stack based return adress overwrites.

To show that this implementation is able to generate exploits for vulnerabilities where a buffer containing the shellcode is mutated after the write of the shellcode, we also tested a vulnerable program that showed this vulnerability class. The analysis of this program did however fail due to an unknown bug during the analysis step in angr.

6.3 Arbitrary write exploitation

As discussed in section 6.1 some parts of the algorithms described in section 5 were impossible to implement using *angr*. In this section we will discuss some of the known exploitation strategies used to build exploits for vulnerabilities in the heap and we will show that they often lead to an arbitrary write or other vulnerability class as described in section 5. Once *angr* supports the required features, it can be shown that the algorithms in section 5 are applicable in realistic situations. A full description and discussion on these attacks can be found in Malloc Des-Maleficarum [3].

6.3.1 *House of Force*

The House of Force exploitation method requires a memory allocation where the size of the allocated memory is determined based upon user input. The next memory allocation will then allocate a section of memory with a symbolic address. If other conditions discussed in [3] are met then this symbolic address can be almost any address in the programs memory. Therefore, if a crashing input satisfies these conditions it should be exploitable using the algorithm described in section 5.5.

6.3.2 *House of Mind*

This attack is also described in [3]. It, just as the previous method, requires a range of conditions to be met. If these conditions are met an arbitrary address gets overwritten with the address of a chunk of the heap. The exploited code can be found in figure 1. If we can control the first bytes of this chunk and its content, then an exploit can be made. This is however not exploitable with the method proposed in this research. This will further be discussed in section 8. In addition to that, mitigations have been implemented in the current version of glibc.

6.3.3 *House of Spirit*

This attack requires a buffer overflow which causes an overwrite of an heap pointer, if all the requirements discussed in [3] are met and a malloc statement is made further down in the program then it will return an chunk with an arbitrary address. Which should, like the House of Force attack, be exploitable by using the algorithm discussed in section 5.5 since this essentially is a symbolic write.

```
1: void _int_free(mstate av, Void_t mem) {
2: .....
3: bck = unsorted_chunks(av);        ▷ This value can be
   controlled by user input
4: fwd = bck->fd;
5: p->bk = bck;
6: p->fd = fwd; ▷ At this point the address of the freed
   chunk gets written to the arbitrary address
7: bck->fd = p;
8: fwd->bk = p;
9: ..... }
```

Figure 1: Code in glibc that leads to arbitrary write

6.3.4 *Unlink*

There was a vulnerability in the unlink macro used by the free method in glibc. CRAX[6] has shown that this kind of vulnerability is exploitable by determining exploitable constraints. However, like the House of Mind attack the current version of glibc is not vulnerable to this attack anymore.

7. DISCUSSION

In this section the results will be discussed in addition to some considerations about the performance and applicability of the tool.

7.1 Performance

It should be noted that although the algorithms in section 5 create correct constraints, a constraint solver might not always be able to solve the constraints because of the complexity that some symbolic states have. This means that although the implementation used to verify the correctness of the algorithms worked on the example programs, that it could hang on more complex and real life applications.

7.2 Arbitrary write

Due to a bug in *angr* we were unable to register the methods called during the symbolic execution of a program. This leads to the inability to find which entry in the general offset table needs to be symbolic to generate an exploit. Therefore, section 5.5 contains a theoretical discussion on whether the algorithm can exploit arbitrary writes.

7.3 Heap exploitation

Unfortunately, at the moment of writing this document, using the tool *angr* we were unable to construct an application that performed analysis on the methods from malloc in glibc. Therefore, it was impossible to validate the algorithms on vulnerabilities that needed key elements from the malloc implementation in glibc to be exploitable. However, there is a discussion in section 6.3 on whether modern exploitation techniques, and whether they resemble an arbitrary write vulnerability.

7.4 Applicability to modern software

Modern systems and software have exploit mitigation systems in place, which prevent some vulnerabilities from being exploitable. This means that to exploit these programs, the exploits need to be more complex to mitigate the mitigation techniques. Exploit mitigation techniques are discussed in section 3.3. This results in the fact that the exploits generated using the algorithms from this research often will often not work. This does not mean that these vulnerabilities are not exploitable by automatic exploitation methods, and is a good point for future research discussed in section 8.2.

8. FURTHER RESEARCH

During the concolic execution of a crashing input all paths that are unreachable with that input remain unexplored, this is the main benefit of concolic execution, where the analysis allows for a detailed understanding of the program state without having the problem of state explosion. There is a downside to this method, because the given input might cause a crash because of a vulnerability, but the taken path can be unexploitable.

Take for example a simple stack based buffer overflow where the input is just long enough to overwrite the first byte of the return address, this will probably not be exploitable, while a minor change in the constraints might allow for exploitation.

In this case a method that identified the instruction that performed the illegal memory action and could efficiently explore the state space to try to re execute that instruction, would allow an automatic exploitation program to find an exploitable path.

Another example vulnerability that can cause an input to crash without following an exploitable path. Is a vulnerability needs to be exploited using the code in malloc, if the input does not satisfy the complex conditions necessary for an arbitrary write then the algorithms in section 5 will not be able to generate an exploit.

It would therefore be interesting to mark some methods like malloc and free to be executed without removing unsatisfiable paths. A tool using this method would not have the problem of state explosion, and should be able to cope with the problem that a minor path change might allow for an exploitable path.

8.1 House of Mind

As stated in section 6.3.2 the method proposed in this research can not yet exploit arbitrary writes where we can not control the expression that is being written. Furthermore, it can also not handle situations where the instruction pointer is not symbolic while the memory at the instruction pointer is. It should be clear that if we place shellcode at this address, that an exploit could be made. This adaption should allow for more vulnerabilities to be exploited using automatic exploitation techniques and is therefore an apt option for further research.

8.2 Hardening

Like noted in the research for Mechanical Shellphish[12] we could use the implementation provided in Q[10] to harden most of the exploits generated by the algorithms in this research. However, in addition to using the methods proposed in that research more hardening methods could be implemented. Another useful topic that could be researched are multi stage exploits. Where the attacker first needs to exploit an information leak before it can build a control flow hijacking exploit. This would probably require broader analysis than concolic execution of a crashing input.

8.3 Other exploitation methods

There are multiple ways that vulnerabilities can be exploited, one of which we have been able to automate. However, this technique is not applicable to every vulnerability. Other methods, like return to libc attacks or exception handler overwrites in addition to other techniques like jump to reg methods that can increase the reliability of exploits can be researched.

8.4 Performance

To avoid complex pseudo code algorithms, we did not look

at the optimization of the exploit generation algorithms. The example programs took about 50 seconds to complete, 90% of this time was taken up by the calculation of all the shellcode constraints. One way that the algorithms can be sped up, is if we immediately check if the instruction pointer can be constrained to the address of the shellcode, instead of when all shellcode constraints have been collected. In addition to that, concurrent calculation of the shellcode constraints should be possible, allowing the utilization of more processor cores.

9. CONTRIBUTIONS

This research focused on generating actionable proof of concept exploits, instead of focusing on the whole process, thus providing an excellent base to compare implementations of exploit generating systems. In addition to that, the research paper discussed the existing exploit generation techniques thoroughly, thus providing an overview of the current state of exploit generation.

Moreover, tools that are used to discover vulnerabilities can be enhanced to use the proposed method. These tools will then be able to recognize security critical bugs, thus assisting developers in prioritizing what bugs should be handled first.

Finally, this research will provide researchers with a base algorithm for further research in this field.

10. CONCLUSION

To guarantee the security of systems it is important to find security vulnerabilities. Part of this is filing bug reports with developers when you have found a bug or vulnerability. However, since the amount of bug reports a development team receives can be more than they are able to resolve, it is important to prioritize security critical bugs. By providing researchers with a detailed analysis of a crash, in addition to a working proof of concept exploit, developers will be able to prioritize security bugs. In this research we looked at methods to automatically generate Proof of Concept exploits to help in this process. We looked at three vulnerability classes, namely pointer overwrites, return address overwrites, and arbitrary writes, and we implemented a tool that could determine an exploit return address overwrite and pointer overwrite vulnerabilities. We also provided theoretical discussion on arbitrary write vulnerabilities, however due to a bug in the supporting framework *angr*, we were unable to implement this in the tool. In section 6.3 we have shown that some heap exploitation techniques can be categorized as arbitrary write exploits. This means that they should be exploitable using the method in section 5.5.

In addition to that, this algorithm has an advantage over the method described by AEG[2] in that it is able to reason about situations where symbolic memory is mutated after it has been written to memory.

In conclusion, we have not yet found a theoretical limit to the use of concolic execution to generate exploits, besides computational problems that could be caused when the analyzed problem imposes a complex set of constraints, and we have shown that it is applicable to the previously described vulnerability classes.

11. REFERENCES

[1] angr. http://angr.io. Accessed: 2017-19-06.

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

[3] blackngel. Malloc des-maleficarum. phrack, 2009. Url: http://phrack.org/issues/66/10.html Accessed: 2017-19-06.

[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[5] GNU. Gnu debugger. https://www.gnu.org/software/gdb/. Accessed: 2017-19-06.

[6] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *SERE*, pages 78–87. IEEE, 2012.

[7] OWASP. Buffer overflow. https://www.owasp.org/index.php/Buffer_Overflow. Accessed: 2017-19-06.

[8] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[9] 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 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[10] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*. USENIX Association, 2011.

[11] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[12] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.