# Analysis of the CRAX Vulnerability Automatic Utilization Process

Zhao Chao[1], Pan Zulie[1], Huang Zhao[1] and Huang Hui[1]
[1]*National University of Defense Technology, Anhui, China*

Keywords: Symbolic execution, Automatic analysis, vulnerability.

Abstract: In recent years, the number of software vulnerabilities has been on the rise, and the harm of software vulnerabilities has become more and more serious. However, there are so many software vulnerabilities that simple manual analysis cannot meet the requirements. In view of the above problems, this paper introduces CRAX. CRAX is a new framework based on symbolic execution. And it is to act as a backend of static/dynamic program analyzers, bug finders, fuzzers, and crash report database. It can automatically, efficiently and quickly analyze software vulnerabilities and generate stable and efficient test cases. The paper analyzes CRAX's automatic analysis process of vulnerability program and the generation process of test cases in detail, and in chapter 3 and chapter 4, it emphatically introduces the constraint construction and reconstruction process in the process of CRAX's automatic analysis.

## 1 INTRODUCTION

Vulnerability is a defect in the specific implementation of hardware, software, protocol or system security policy, which enables an attacker to access or break the system without authorization (Yaquan, 2016). In recent years, the number of software vulnerabilities has been on the rise. The malicious Web attack events that software vulnerabilities should be launched violate citizens' rights and interests, spread a wide range of computer viruses, cause major economic losses, and implement advanced sustainable attacks to trigger national security incidents (McGraw, 2016). Therefore, software vulnerability analysis has become a problem that cannot be ignored in the computer field.

When the program crashes, the traditional analysis is to manually analyze the availability of crash so that it can determine whether the crash is caused by internal logic errors or by external input. If it is caused by external input, then it is likely to be a very serious crash or even an exploitable vulnerability. However, due to the numerous software crashes, pure manual analysis has been unable to meet the requirements, so how to quickly and efficiently analyze the availability of crash has become one of the key issues in the field of vulnerability mining and analysis.

## 2 CRAX

At present, with the continuous development of automatic program analysis, especially the introduction of symbol execution, stain analysis and other technologies into the fields of software crash analysis and vulnerability mining, various software vulnerability utilization automatic construction technologies have been proposed (Liang and Purui, 2016). At IEEE S&P conference in 2008, D.Brumley et al first proposed AEPG based on patch comparison (Brumley and Poosankam, 2008). At NDSS conference in 2011, T.Avgerinos et al first proposed AEG based on source code analysis (Avgerinos and Cha, 2011). On the basis of AEG technology, in 2012 IEEE conference, Huang Shikun et al proposed the automation framework CRAX based on AEG method improvement.

Based on S2E (Chipounov and Kuznetsov, 2011) environment model, KLEE (Cadar and Dunbar, 2008) symbol virtual machine and QEMU (Bellard, 2005) processing simulator, this framework is a new platform for symbol execution. To generate control flow hijack attacks, CRAX focuses on symbolized EIP, registers, and Pointers. A systematic method is



Chao, Z., Zulie, P., Zhao, H. and Hui, H.
Analysis of the CRAX Vulnerability Automatic Utilization Process.
DOI: 10.5220/0008097100320037
In *Proceedings of the International Conference on Advances in Computer Technology, Information Science and Communications (CTISC 2019)*, pages 32-37
ISBN: 978-989-758-357-5

proposed to search the maximum continuous symbol memory for payload injection.

## 3 BUILD CONSTRAINT

In the automatic analysis and utilization of the program, CRAX first determines the availability of the program. If the EIP of the program can be overwritten, the risk of the program is determined. If the determination program has available risks, CRAX transfer to the available constraint building and solving process. In the process of building a constraint, CRAX first looks for a large section of the continuous symbolic area, and then tries to construct an input which can cause overflow and hijack the control flow. Through input, CRAX can build shellcode constraints、nop constraints and eip constraints. After successful construction, CRAX combine shellcode constraints, eip constraints, nop constraints, and the path constraints, if the result is true, then it indicates that the constraint is constructed successfully, and the code distributes after the constraint is constructed successfully is shown in figure 1. The symbolic area grows from low address to high address, and shellcode is arranged at the bottom of the symbolized area:

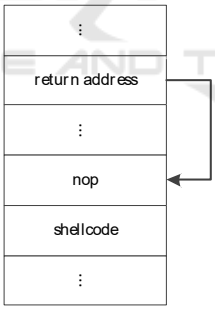


Figure 1: The code distribution after a successful constraint building.

1. Build shellcode constraint: Overlay a contiguous symbolic area with the specified shellcode that implements specific functions.
2. Build nop constraint: Arrange a large section of nop in front of shellcode so that it doesn't affect shellcode's functionality, meanwhile it can increases the chances that the IP registers jump to shellcode.
3. Build eip constraint: Overwrite the value of the IP register as the specified value, so that EIP can jump directly to the nop area or to the start address of shellcode after the program flow hijacking is successful. When the vulnerability is triggered and the program flow is hijacked, the program can jump to the nop area and execute a long section of nop and then execute shellcode, or jump directly to the shellcode starting address to execute shellcode, thus completing the specified function of the attacker.
4. Path constraint: The condition required for memory placement to be met when the specified location is overridden by the input to the specified code.

Therefore, CRAX constructed shellcode constraint, nop constraint, and eip constraint on the shellcode area, nop area, and the affected IP register area, and respectively combined the three constraints with path constraint to form a harness code constraint named exploit constraint that directly overrides the return address-type vulnerability utilization with a fixed address.

Only when there is a solution to the exploit constraint, the availability of the target program is considered to be true, and further constraint simplification and constraint solving are conducted to obtain the target program utilizing code. The specific construction process is shown in figure 2:

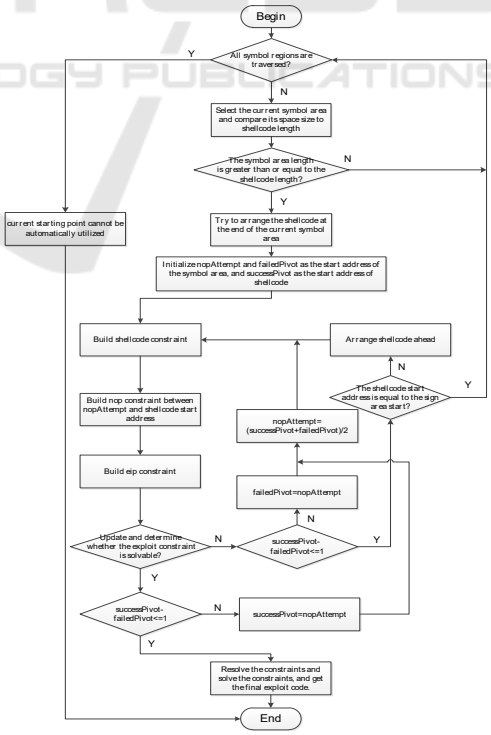


Figure 2: Constraints construct flow diagrams.

## 3.1 Shellcode Constraint

The shellcode constraint is used to constrain the arrangement of shellcode in the symbolized area. In order to find an appropriate size memory area in the memory space, CRAX finds areas that can be affected by external input and records these continuous symbol memory areas by symbolizing the external input and by tracking the data flow. In these continuous symbolic memory areas, the contents of a specified memory area can be overwritten as long as a specific data stream is built as an external input. Then, in each continuous memory area, CRAX writes the shellcode of specific function into memory byte by byte and builds the shellcode constraint until it succeeds in a certain memory area. The ultimate result of constraint solving is to write the shellcode for specific functions in the specified symbol area through external input.

Before constructing the shellcode constraint in the symbol memory area, the relationship between the size of the current symbol memory area and the length of the shellcode to be written is first compared. If the size of the current symbol memory area is less than the length of shellcode, it indicates that the current symbol memory area is too small to arrange the shellcode to be written. It means the shellcode constraint builds failed this time, and need to look for new symbol memory areas for the next constraint build attempt; If the current symbol memory area is greater than the length of shellcode, it indicates that the symbol memory area is large enough for shellcode to further try the shellcode constraint construction.

When building shellcode constraint, CRAX places the shellcode at the end of the symbolic memory area.

After determining the area used to arrange shellcode, the constraint is constructed word by word, forming a constraint to determine whether the value of the corresponding memory area used to store shellcode can be equal to the byte at the corresponding offset in shellcode. The specific shellcode constraint build process is shown in table 1:

Table 1: The process of building shellcode constraint.

```
if symbolicArea_size < shellcode_size
Find new symbolic area
for i = 0 to shellcode_size do
shellcode_const=EqExpr(readMemory[shellcode
_Addr+i], shellcode[i])
```

## 3.2 Nop Constraint

The nop constraint is the constraint used to populate the nop instruction value between the shellcode starting address and the start address of the symbol memory area. Nop instructions are "empty instructions," meaning that the execution of an instruction does not change the register accessible to any program, the processor status flag, or main memory. The machine code is 0x90 under x86 conditions. Therefore, nop instruction does not produce any actual execution effect except that it takes up some memory and requires little execution time. In the process of constructing the constraint, the main purpose of building the nop constraint is to increase the coverage of nop, so that the modified IP register can jump to the area domain with greater probability, and then to the shellcode area.

The introduction of the nop constraint is mainly to make the value of the overriding return address not necessarily a fixed value of the shellcode start address, but any value between the start address of the nop area and the start address of the shellcode start address, so that the shellcode can eventually be executed. In this way, the flexibility of code utilization is improved, the value range of the returned address is improved, and the possibility of solving eip constraint is also improved.

In order to maximize the range of data values used to cover the returned addresses, we need to start from the shellcode starting address, to the start address of the symbol memory area, and arrange the continuous nop area as much as possible. For this reason, CRAX tries to arrange the largest nop area using dichotomy before the shellcode start address, from the start address of the current symbol area to the start address of the shellcode. Then CRAX constructs the constraint byte by byte which determines whether the nop block can be equal to "\x90", and then combine to form the corresponding nop constraint. The nop constraint build process is shown in table 2:

Table 2: The process of building nop constraint.

```
for i = 0 to shellcode_Addr – nopAttempt do
nop_const = EqExpr(readMeMory[nopAttempt + i] ,0x90 )
if(Query(path_const ,exploite_const )){
successPivot = nopAttempt
nopAttempt = (faildPivot + successPivot)/2
if(isPivotsOverlapped(failedPivot ,successPivot))
        return nop_const
```

```
else
        build nop_const based on new parameters
else{
failedPivot = nopAttempt
nopAttempt = (failedPivot + successPviot)/2
if(isPivotsOverlapped(failedPivot ,successPivot))
        return nop_const = 0
else
        build nop_const based on new parameters
    }
}
```

### 3.3 Eip Constraint

The eip constraint is the constraint on the IP register value when the control flow hijacking occurs. Since the value of the IP register is the address of the next instruction to be executed by the CPU, a constraint needs to be built to form a constraint on the value stored in the IP register after the control flow hijacking, so that the value points to any address between the start address of the nop area and the start address of shellcode. After the program returns, the program flow will jump to the nop area or directly to the shellcode starting address for execution.

The eip constraint build process is shown in table 3:

Table 3: The process of building eip constraint.

```
lowerBound = nopAttempt
upperBound = shellcode_Addr
eipConstraint = AndExpr (UgeExpr ( m_eipValue ,lowerBound),
UleExpr(m_eipValue ,upperBound ));
```

## 4 CONSTRAINT REBUILDING

The exploitability constraints built on shellcode constraint、nop constraint and eip constraint may fail to combine with the path constraint due to bad characters or conflict with the path constraint. However the current constraint solving failure does not necessarily mean the automatic generation of utilizing code failure. The fact that no solution for the current constraint actually represents the kind of data layout corresponding to the current constraint is not feasible. It is usually caused by conflict with the current path constraint and other reasons. Therefore, it can maintain the original function and avoid conflict with the current path constraint by adjusting the arrangement of data, so as to obtain the final use code.

In fact, CRAX expects to automatically get a utilization code that can jump directly to shellcode through the nop area and give the largest possible nop area in the final utilization code, that is, as many nop empty instruction values as possible before the shellcode without conflict with other constraints. Therefore, it is necessary to adjust the constraint after the construction of the constraint and judge the resolvability after the reconstruction. This process of adjusting and rebuilding constraints is called constraint rebuilding.

### 4.1 Rebuild Shellcode Constraint

For the shellcode constraint, after the construction of the constraint is completed, if the solution of the exploit constraint fails, since the initial shellcode is started at the end of the current symbol memory area, then consider adjusting the position of shellcode constraint when constructing the shellcode constraint. If the starting position of shellcode is not equal to the starting position of the current symbol area, move shellcode forward by a byte and rebuild the other constraint until the exploit constraint has a solution; If the shellcode start location has been aligned with the start address of the current symbol area, but is still unsolvable, the current symbol memory area is determined to be unavailable. Try to find a new symbol memory area and start constructing the constraint again until it is solvable and available code; If all symbol memory areas are tried and still unsolved, the current program is determined not to be automatically utilized

### 4.2 Rebuild Nop Constraint

For the nop constraint, in order to make the nop area in the code as large as possible, CRAX uses dichotomy to gradually determine the nop area step by step between the start address of the symbolic memory area and the start address of shellcode.

1. When the constraint is first constructed, try to arrange all the nop between the symbolic memory area and the shellcode starting address, that is, the initial before endpoint and after endpoint used in the record dichotomy are low= symbolicArea _startAddr, high= shellcodeaddress-1, and the nop area starting address is nopAttempt= symbolicArea _startAddr.

2. Try to construct a constraint that sets all contents from nopAttempt to high as nop, and add nop constraint to exploit constraint. After building exploit constraint, determining the solubility of the result with the combination of exploit constraint and path constraint.

3. If the decision result is solvable, it indicates that under the current condition, all nop areas can be covered as nop. Try to use dichotomy to expand nop interval, so determine whether abs(high-low) <= 1. If not, the nop area still can expand. Make successPivot = nopAttempt, nopAttempt = (failedPivot + successPivot)/2, and then go to step 2. If true, the nop area cannot be expanded, then nop_constraint is returned.

4. If the decision result is unsolved, in the current condition, the number of bytes in the nop area cannot be assigned to the nop, so try to narrow down the nop interval by using the dichotomy. Determine whether the condition abs(high - low)<=1 is true. If not, make low=nopAttempt, nopAttempt=(low+ high)/2, and then go to step 2 to construct the nop constraint. If the condition is true, it returns nop_constrait 0.

### 4.3 Rebuild Eip Constraint

1. For eip constraint, if the nop constraint is not empty, try to make the value used to cover the ip register within the nop area, or point directly to the shellcode starting address, and construct the corresponding constraint accordingly. If the nop constraint is empty, try to directly builds constraints that make the ip register equal to the shellcode start address.

2. If eip constraint is added to exploit constraint determine whether there is a solution to the exploit constraint. If yes, try to determine the maximum range that nop area can obtain by dichotomy, that is to advance the start address of nop area by dichotomy. Make high=nopAttempt, nopAttempt=(low+high)/2, and determine whether the end of dichotomy condition abs(high-low) <= 1 is true. If it is true, then the exploit constraint is the final constraint and constraint building has solution; then simplifing the constraint and solving the constraint by the constraint solver, the final vulnerability exploit constraint code can be obtained. If the end condition of dichotomy is not true, try again to construct the nop constraint in 2 according to the new high, low and nopAttempt values.

3. If the exploit constraint obtained in 2 is determined to be unsolved after solving, it means that the eip constraint cannot fall within the nop area or directly points to the shellcode starting address. Then, it is considered that whether the nop area space can be expanded by dichotomy to expand the feasible range of eip constraint. Then, the start address of nop area is brought forward to make high=nopAttempt, nopAttempt=(low+high)/2, and then judge whether the condition abs(high-low)<= 1 is true. If it is not true, the nop constraint is reconstructed again according to the new high, low and nopAttempt values. If true, when reaching end but there is no feasible solution, go to step 1, move the shellcode start address one ahead, and rebuild the shellcode constraint.

## 5 CONCLUSION

CRAX is an automatic development and generation framework based on S2E. In order to generate control flow hijacking attacks, it focuses on symbolized EIP, registers, and pointers, and propose a systematic method for searching maximum contiguous symbolic memory for payload injection.

Experiments on various vulnerable sample codes show that CRAX can handle different types of control flow hijacking vulnerabilities. At the same time, it analyzed and utilized the vulnerability faster and more efficiently than manual debugging. CRAX is also a viable and powerful development tool for real-world environments.

But CRAX has limitations. First of all, it mainly focuses on the automated analysis of stack overflow vulnerability, while the automated analysis of heap overflow vulnerability remains to be studied. Secondly, it does not consider the influence of ASLR, DEP and other protection mechanisms on automatic analysis.

## REFERENCES

Lin Yaquan. , 2016. War on Vulnerability. Beijing: Electronic Industry Press.

McGraw, G. , 2016. Software security: Building Security In. Boston, MA: Addtion-Wesley Professional.

Liang, H. , Purui, S. , 2016. Research Progress of Software Vulnerability Automatic Utilization. China Education Network.

Brumley, D. , Poosankam, P. , Song, D. X. , & Zheng, J. . 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. IEEE Symposium on Security & Privacy. IEEE.

Avgerinos, T. , Cha, S, K. , Tao, B, L, T. , Brumley, D. , 2011. AEG:Automatic Exploit Generation. In Proceedings of the Networkand Distributed System Security Symposium (NDSS).

Chipounov, V. , Kuznetsov, V. , Candea, G. , 2011. “S2E: a platformfor in-vivo multi-path analysis of software systems,” in Proceedings of the 16th International Conference on ArchitecturalSupport for Programming Languages and Operating Systems(ASPLOS’11), Newport Beach, CA, USA.

Cadar, C. , Dunbar, D. , Engler, D, R. 2008. “KLEE: Unassistedand Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in Proceedings of the 8th USENIXSymposium on Operating Systems Design and Implementation(OSDI’08), San Diego, California, USA.

Bellard, F. , 2005. “QEMU, a fast and portable dynamic translator,”in Proceedings of the FREENIX Track: 2005 USENIX AnnualTechnical Conference, Anaheim, CA, USA.