CPU Exhaustion attack String generation from CCFGs using LPs and ILPs

Rajarshi Basak

December 2016

1 Introduction

An application-level security-level attack exploits weaknesses in an application's code. Among several known application-level security attacks, we are interested in two specific types of attacks which exploit loops in network applications: CPU exhaustion attacks and buffer overflow attacks.
A CPU exhaustion attack is a special case of a Denial of Service (DoS) attack, which allows malicious users to gain control over a system either locally or remotely across a network, denying access to other legitimate users in turn. These attacks can be launched by constructing inputs that cause resource utilization to climax during their processing.
Buffer-overflow attacks, meanwhile, permit attackers to destabilize applications. A buffer-overflow attack is launched by constructing inputs that store large data in a stack buffer of size less than the usual size, hence corrupting the execution stack of an application. Although there are several ways to launch buffer-overflow attacks, we focus on the ones that can be launched via exploiting loops.

1.1 Loop-exploiting attacks

A loop-exploiting attack is launched by a valid string input that bypasses sanitization checks in the application, which in turn causes an exponential number of loop iterations with respect to the length of the concerned string input. It has been observed that these attack-causing inputs follow specific patterns and include correlations among characters in the input, which help execute paths that lead to recursive calls or nested loops.

2 Formal Definition of Loop-exploiting attacks

The formal definition for the problem of generating loop-exploiting attacks is as follows.

Assuming that the application under analysis accepts string inputs, for example $s_1, s_2, ..., s_n \in$ regular language, RL, and includes vulnerabilities for causing loop-exploiting attacks, the general problem is to generate a likely attack-causing input s_i, where $s_i \in RL$ and satisfies the requirement specific to the attack under analysis. If $s_i \in RL$, the string inputs are valid inputs and bypass sanitisation checks.

2.1 CPU Exhaustion attacks

Given that an application under analysis accepts a single input u_i, there are N statements in the application and each statement s_i in the application takes a constant amount of CPU time,

t for executing that statement, let $ET(u_i)$ represent the execution time of the application with the input u_i. Then we have

$$ET(u_i) = \sum_{i=1}^{N} c_{s_i} t \tag{1}$$

where c_{s_i} represents the number of times the statement s_i is executed.

An input u_i is then classified as a likely attack causing input if the application execution with that input causes $ET(u_i) > ET-THRESHOLD$ where $ET-THRESHOLD$ is a user-defined threshold value indicating the minimum amount of time the application should be executed to classify the input as a likely-attack causing input.

3 Context-free grammars and Context-free languages

A Context-free grammar(CFG) describes a certain type of formal grammar; it is essentially a set of production rules (replacements) that describe all possible strings in a given formal language.

For example, the rule

$$A \rightarrow \alpha$$

substitutes A with a. It is possible to have multiple replacement rules for a given value, i.e.

$$A \rightarrow \alpha$$

$$A \rightarrow \beta$$

implies A can be replaced with either α or β.

All rules in a CFG are either one-to-one, one-to-many, or one-to-none. The left-hand side of a production rule is always a non-terminal symbol, meaning that the symbol does not appear in the resulting formal language. This means that in this case the language contains the letters α and β but not A. An example of a context-free grammar that describes all two-letter strings containing the letters α and β is:

$$S \rightarrow AA$$

$$A \rightarrow \alpha$$

$$A \rightarrow \beta$$

Starting with the non-terminal symbol S, we use $S \rightarrow AA$ to turn S to AA, following which if we apply $A \rightarrow \beta$ to the first A, we get βA. If we then apply $A \rightarrow \alpha$ to the second A we get $\beta\alpha$. If instead we apply $A \rightarrow \alpha$ to the first A, followed by $A \rightarrow \beta$ to the second A, we get $\alpha\beta$. In other words, the same process can be employed, in which the second two rules are applied in different orders to get all possible strings within this simple CFG example.

Since both α and β are terminal symbols and in CFGs terminal symbols are not allowed to appear on the left-hand side of a production rule, we cannot apply any more rules to this example.

A constrained Context-free grammar (cCFG) is a special CFG in which the production rules are constrained by upper (and sometimes lower bounds). In other words, the $i_t h$ production rule has a frequency f_i, which means that we can apply the rule a maximum i number of times to generate the language.

The languages generated by CFGs are called Context-free languages(CFL). It is to be noted that different CFGs can generate the same context-free language(CFL).

4 Using LPs and ILPs to generate attack causing strings from cCFGs

In this project, we are given a particular constrained Context-free grammar, and our objective is to use it to generate the attack causing inputs(string). The input string here is simply the string of maximum length. We achieve this by first re-expressing the cCFG problem from the Network flow point of view, and then solving the Network-flow equivalent problem using linear programming (solving a linear program to find out the frequencies of each of the production rules that maximizes the string length). This is because any network-flow problem can be reduced to a linear program. Next we solve an integer-linear program to resolve the frequency order, i.e. the order in which the production rules should be applied to achieve the string of maximum length whose production rule frequency values were obtained by solving the previous linear program. Finally, we reconstruct the string (of maximum length) using the solutions of the linear program which solves the production rule frequency value problem (PRFVP) and the integer-linear program which solves the production rule frequency order problem (PRFOP).

It is to be noted here that the constraints of maximum flow problem yields a coefficient matrix with the properties that ensures the matrix is totally unimodular. Thus, such network flow problems with bounded integer capacities have an integral optimal value. In light of this, we need not ensure that the PRFVP problem is solved using an integer-linear program; a simple linear program instead would provide integer solutions.

To illustrate how this procedure works, we take a cCFG example and apply the steps mentioned above to solve the production rule frequency solution and the production rule order solution.

Let the cCFG be the following:

$S \to AB$ with fm_1; Rule R_1
$A \to aB$ with fm_2; Rule R_2
$B \to b$ with fm_3; Rule R_3

where fm_1, fm_2, fm_3, represent the upper bounds on the frequencies of the production rules R_1, R_2 and R_3 respectively.

$f_1 = 2$, $f_2 = 2$, $f_3 = 4$.

Let f_1, f_2 and f_3 be the number of times the rules R_1, R_2 and R_3 need to be applied respectively to generate the maximum length string.

5 The Network-flow problem : Reduction to a linear program

Given a network consisting of a directed graph $G = (V, E)$ and two special nodes $s, t \in V$, which are the source and target nodes respectively, and capacities (maximum allowed flow) $c_e > 0$ on the edges, we are interested in sending maximum possible quantity of oil without exceeding the capacities of any of the edges. A particular shipping scheme is referred to as a flow and consists of the variable f_e for each edge e of the network, satisfying the following two properties:

(1) The flow does not exceed the edge capacities: $0 \le f_e \le c_e$ for all $e \in E$.

(2) The amount of flow entering node u equals the amount leaving u for all nodes except s and t, i.e.

$$\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{u,z} \tag{2}$$

Also the size of the flow is the total quantity sent from s to t, and by the conservation principle, is equal to the quantity leaving s:

$$size(f) = \sum_{(s,u)\in E} f_{s,u} \tag{3}$$

Thus the goal is to assign values to $(f_e : e \in E)$ that will satisfy a set of linear constraints and maximize a linear objective function, and this is a linear program. Hence the maximum-flow problem reduces to linear programming.

The next step is to show that the problem of finding the frequency values in our cCFG that maximizes the length of the string can be converted to a network-flow problem.

5.1 Converting the PRFVP to a network-flow problem and solving the concomitant LP

To convert the PRFVP to a network-flow problem, we first note that the in-flow of each non-terminal should equal the outflow of the same. Thus, we have the following constraints:

$$1 = f_1 \tag{4}$$

$$f_1 = f_2 \tag{5}$$

$$f_1 + f_2 = f_3 \tag{6}$$

implying that the inflow should equal the out-flow for the non-terminals S, A and B respectively. This is equivalent to amount of flow entering node u equals the amount leaving u for all nodes except s and t in the network-flow problem.

The next set of equations are the constraints for the bounds on the number of times (frequencies) we can apply the production rules. This is equivalent to the flow not exceeding the edge capacity in each of the network flow problem.

$$0 \le f_1 \le 2 \tag{7}$$

$$0 \le f_2 \le 2 \tag{8}$$

$$0 \le f_3 \le 4 \tag{9}$$

Once we have assigned weights to the terminals (a and b),
$w(a) = 1$ for the terminal a, and $w(b) = 1$ for the terminal b
we can define the objective function that we are trying to maximize, which is

$$w^* = w(a)f_2 + w(b)f_3 = f_2 + f_3 \tag{10}$$

as only rules R_2 and R_3 lead to terminals (with frequencies f_2 and f_3 respectively).

Solving the linear program using Mosek (a java linear programing extension), we have the solution for the production rule frequency value problem:

$f_1 = 2$, $f_2 = 1$, $f_3 = 2$ and the objective function value is $w^* = 3.0$.

Here we not that $F = \sum_i f_i = f_1 + f_2 + f_3 = 4$.

6 Solving the PRFOP using an ILP

Even though we have the solution for the production rule frequency value problem (PRFVP), we still need to know the order in which to apply these production rules to generate the required string as different orderings would generate different strings (of different lengths). Hence we need to solve the production rule frequency order problem (PRFOP) next, using an integer-linear program.

In order to frame the ILP, we first note that the states and the production rules can be represented by vectors (of dimension = number of non-terminals). Since there are 3 non-terminals in the cCFG (S, A and B), the vectors would be of dimension 3.

For production rule R_1, which transforms S to AB, it can be easily seen that the vector for the starting state S is $(1, 0, 0)$, for the rule R_1 is $(-1, 1, 1)$ (as it converts a non-terminal state S to a non-terminal state AB, containing zero S, one A, and one B), and for the leading state AB is $((0, 1, 1))$.

Thus

$$S = (1, 0, 0) \tag{11}$$

$$R_1 = (0, 1, 0) \tag{12}$$

Similarly, we have

$$R_2 = (0, -1, 1) \tag{13}$$

$$R_3 = (0, 0, -1) \tag{14}$$

Noting from the previous LP solution that $F = \sum_i f_i = 4$, we conclude that we can apply the production rules $R_1 ... R_3$ a maximum 4 times starting from the initial state S given by $(1, 0, 0)$. But the final state has to be $(0, 0, 0)$ since it won't have any non-terminals and only terminals. Thus we have $4 - 1 = 3$ intermediate unknown states.

We define a new variable r_i, given by

$$r_{i,j} = 1 \tag{15}$$

if rule R_i is to be applied at stage j, and

$$r_{i,j} = 0 \tag{16}$$

if rule R_i is not to be applied at stage j.

Since at each stage, we can apply only one rule from the three possible rules, and there are 4 stages or transitions ($j = 1$ to $j = 4$) (excluding the initial stage), we have the following set ($F = 4$, $i = 0$ to $i = 3$) of constraints:

$$\sum_{j=1}^{R=3} r_{ij} = 1 \tag{17}$$

which, when expanded, yields

$$r_{01} + r_{02} + r_{03} = 1 \tag{18}$$

$$r_{11} + r_{12} + r_{13} = 1 \tag{19}$$

$$r_{21} + r_{22} + r_{23} = 1 \tag{20}$$

$$r_{31} + r_{32} + r_{33} = 1 \tag{21}$$

As there are F transitions, we have $F = 4$ such constraints.

Again, since the solution from PRFVP informs us the number of times we can apply each rule R_i in these 4 steps, we have the following set ($R = 3$, $j = 1$ to $j = 3$) of constraints

$$\sum_{i=0}^{F=14-1=3} r_{ij} = f_i \tag{22}$$

which when expanded, yields

$$r_{01} + r_{11} + r_{21} + r_{31} = 1 \tag{23}$$

$$r_{02} + r_{12} + r_{22} + r_{32} = 1 \tag{24}$$

$$r_{03} + r_{13} + r_{23} + r_{33} = 2 \tag{25}$$

Since there are R such rules, we have $R = 3$ such constraints.

Finally, we have the constraints that capture the transitions between each of the adjacent stages.

If the starting state is $S(1, 0, 0) = X_0(x_{01}, x_{02}, x_{03})$, then after the first transition, the second state is given by $X_1(x_{11}, x_{12}, x_{13})$, and we have the transition rule constraint

$$X_1(x_{11}, x_{12}, x_{13}) = X_0(x_{01}, x_{02}, x_{03}) + r_{01}(-1, 1, 1) + r_{02}(0, -1, 1) + r_{03}(0, 0, -1) \tag{26}$$

which actually refers to three equations/constraints:

$$x_{11} = 1 - r_{01} \tag{27}$$

$$x_{12} = 0 + r_{01} - r_{02} \tag{28}$$

$$x_{13} = 0 + r_{01} + r_{02} - r_{03} \tag{29}$$

Similarly, for the second transition, the relation between the second state and the third state is given by

$$X_2(x_{21}, x_{22}, x_{23}) = X_1(x_{11}, x_{12}, x_{13}) + r_{11}(-1, 1, 1) + r_{12}(0, -1, 1) + r_{13}(0, 0, -1) \tag{30}$$

which refers to the three constraints

$$x_{21} = x_{11} - r_{11} \tag{31}$$

$$x_{22} = x_{12} + r_{11} - r_{12} \tag{32}$$

$$x_{23} = x_{13} + r_{11} + r_{12} - r_{13} \tag{33}$$

Hence in general, for the jth transition, we have,

$$X_j(x_{j1}, x_{j2}, x_{j3}) = X_{j-1}(x_{(j-1)1}, x_{(j-1)2}, x_{(j-1)3}) + r_{(j-1)1}(-1, 1, 1) + r_{(j-1)2}(0, -1, 1) + r_{(j-1)3}(0, 0, -1) \tag{34}$$

which again can be split into three constraints.

Since there F transitions, we have $F * 3 = 4 * 3 = 12$ such constraints.

Thus we have $F + R + 3F = 19$ constraints.

As far as the number of variables in concerned, it is easy to note that we have $(F - 1)$ variables, x_{ij}'s, for the state vectors (in this case 9), and $3R$ (12 in this case) variables, r_{ij}'s, for the production rule choices.

Further, we have the following bounds for the variables, the r_{ij}s have to be either 0 or 1, and the x_{ij}s have to be non-negative integers (since no state can have a negative number of terminals or non-terminals).

Since we are interested in the order of applying the rules, this is a feasibility problem, and the objective function is a set of zeros; i.e. it has zero coefficients for each of the variables.

6.1 The output

Now that we have defined the variables, the constraints, the bounds on the variables and the objective function, we can feed this data to our integer-linear programming solver package in java (mosek).

The optimal solution obtained is as follows:

```
    x11: 0.000000000
x12: 1.000000000
x13: 1.000000000
x21: 0.000000000
x22: 0.000000000
x23: 2.000000000
x31: 0.000000000
x32: 0.000000000
x33: 1.000000000
r01: 1.000000000
r02: 0.000000000
r03: 0.000000000
```

```
r11: 0.000000000
r12: 1.000000000
r13: 0.000000000
r21: 0.000000000
r22: 0.000000000
r23: 1.000000000
r31: 0.000000000
r32: 0.000000000
r33: 1.000000000
```

7 Reconstruction of the string

From the solution we can read off the construction of the string starting from $S(1,0,0)$.

$S(1,0,0) \xrightarrow[(-1,1,1)]{R_1} AB(0,1,1) \xrightarrow[(0,-1,1)]{R_2} aBB(0,0,2) \xrightarrow[(0,0,1)]{R_3} abB(0,0,1) \xrightarrow[(0,0,1)]{R_3} abb(0,0,0)$

Thus the maximum length string in this case is abb (of length $= 3$, as predicted by the PRFVP earlier) which is represented by the vector $(0,0,0)$ and it is obtained after applying a production rule from the three possible rules 4 times.

8 Further directions

In the future, we plan to build two tools: (1) which can construct the constraints for both the PRFVP and PRFOP problems, given a cCFG, and can solve the linear program and integer-linear program, respectively, and (2) which can construct the string from the solutions of the above linear programs.

9 References

(1) Thummalapenta, Suresh, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Tao Xie. "Lexar: Generating String Inputs for Loop-Exploiting Attacks via Evolutionary Techniques."

(2) https://en.wikipedia.org/wiki/Context-free$_grammar$

(3) https://en.wikipedia.org/wiki/Unimodular$_matrixTotal_unimodularity$

(4) Algorithms, 1st Edition by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani, McGraw-Hill Education

10 Notes for code

(1) Lines 45-133 for ccfg4 was borrowed from the mosek library.
(2) Lines 190-332 for ccfg4order was borrowed from the mosek library.