Adaptive Defense Against Adverserial Artificial Intelligence at the Edge of the Cloud using Evolutionary Algorithms

by

Sofiane Djeffal

M.S., Boston University (2011)
B.S., Boston University (2008)

Submitted to the System Design and Management Program
in partial fulfillment of the requirements for the degree of

Master of Science in Engineering and Management

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Author . . . .
System Design and Management Program
May 8, 2020

Certified by . . . .
Una-May O'Reilley
Principal Research Scientist
Thesis Supervisor

Certified by . . . .
Erik Hemberg
Research Scientist
Thesis Supervisor

Accepted by . . . .
Joan Rubin
Executive Director, System Design and Management Program

THIS PAGE INTENTIONALLY LEFT BLANK

Adaptive Defense Against Adverserial Artificial Intelligence at the Edge of the Cloud using Evolutionary Algorithms

by

Sofiane Djeffal

Submitted to the System Design and Management Program
on May 8, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science in Engineering and Management

Abstract

While moving to the cloud increases flexibility for many organisations around the world, it also presents its own set of operational risks and complexities when it comes to keeping data and workflows secure. As data becomes digitized, it is becoming more fruitful for bad actors to try to engage in data theft or disrupt online services for their financial gain, corporate espionage, or general intent to disrupt a service. Computers are also becoming more powerful and sophisticated than ever, allowing them to brute force what were once considered top of the line cryptographic ciphers and algorithms in no time. The cost of protecting an infrastructure is increasing both financially and in terms of human resources needed to support a system's security. Companies are relying on the cloud to provide that protection, and one of the ways the cloud provides it is through Edge nodes that sit in front of their infrastruture. Edge nodes are the first line of defense against threats to a web application.

This thesis explores a new heuristic for approaching threat generation and detection in a network. It aims to demonstrate that with a proper grammar definition along with a strategy, and a reward system, a genetic algorithm can perform better than the existing rulebased system used to generate and defend against a wide breadth of attacks.

This proposes solution focuses on three types of attacks: Data Exfiltration, Server Hijack, and Denial of Service. The goal is to demonstrate that computationally searching for vulnerabilities does not scale well with a rule based system while a genetic algorithm can handle an increase of breadth in attacks with more elegance and better results.

Thesis Supervisor: Una-May O'Reilley
Title: Principal Research Scientist

Thesis Supervisor: Erik Hemberg
Title: Research Scientist

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

First and foremost I would like to thank my parents Mahbouba and Mahieddine for their unconditional love and support, and for being the best role models a son can ask for. My little sister Manel, who challenged me in childhood and inpsires me in adulthood. My girlfriend Manon who has been a pillar of support during the final stretches of my academic journey with her warm presence that brings joy and light to my life every single day. My advisors Erik, Una-May and the rest of ALFA research group for providing invaluable guidance, feedback, and an array of interesting topics to learn from during our bi-weekly lunches. To my SDM cohort and SDM leadership for providing me the venue to challenge myself and for exposing me the a unique set of perspectives I would have never been exposed to outside the program. I hope that I was able to do the same. To my colleagues at Microsoft whose passion, expertise and work ethic continue to be an inspiration every day.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction 17

1.1 Motivation . . . 17

1.2 Research Goal . . . 19

1.2.1 Research Questions . . . 19

1.2.2 Research Approach . . . 20

1.3 Thesis Structure . . . 20

2 Related Work 21

3 Literature Review 23

3.1 Cloud Infrastructure . . . 23

3.1.1 Web Application Servers . . . 24

3.1.2 Edge Computing and BGP Networks . . . 27

3.2 The Web Application Firewall . . . 29

3.3 Grammatical Evolution . . . 31

3.4 Cyber attacks and Threat model . . . 32

3.4.1 Threat Model . . . 32

3.4.2 Advanced Perisitant Threat (ATP) and Cyber Killchain . . . 34

3.4.3 Types of attacks studied in this research . . . 36

4 Case study: Equifax - An HTTP Header Attack 41

4.1 How it happened . . . 41

4.2 Impact and Cost . . . 42

4.3 Lessons learned . . . 42

5 Methodology **43**

5.1 Attacker-Defender system setup . . . 43

5.1.1 The Attacker: A client running a grammatical evolution algorithm . . . 44

5.1.2 The Defender: An application server behind an Edge node running a Web Application Firewall . . . 48

5.2 Methodology Limitations . . . 50

5.3 Parameters tested in Grammatical Evolution . . . 51

5.4 Attack selection and Strategy . . . 52

5.5 Search space . . . 53

5.6 Measure of fitness . . . 54

6 Experiments **57**

6.1 Proof of Concept . . . 58

6.2 Denial of Service: Exploiting an expensive operation . . . 59

6.2.1 Random Search . . . 60

6.2.2 Grammatical Evolution . . . 60

6.3 Increasing the breadth of attacks and strategies . . . 62

6.3.1 The stealthy attacker strategy . . . 63

6.3.2 The persistant attacker strategy . . . 64

6.4 Rescoping the grammar from HTTP Requests to attack missions . . . 66

6.5 Summary of results . . . 69

7 Conclusion and future work **71**

7.1 Limitations . . . 71

7.2 Future Work . . . 72

A Threat Model **73**

A.1 Client to Edge Request . . . 73

A.2 Web API to Database . . . 75

A.3 Edge to Web API Request 78
A.4 Web API to Database Response 81
A.5 Client to Edge Response 84

B Listings **85**

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

1-1 The average and standard deviation of critical parameters 18

3-1 Summary of key differences of different application server infrastructures [1]. 24
3-2 Each subnetwork is called an Autonomous System (AS) and is a large pool of routers run by a single organization, typically an Internet Service Provider [2] 27
3-3 This Figure shows 6 AS. If AS1 need to go to AS3, It has 2 different options [2] 28
3-4 An Edge node sits between the user and the service and provides content delivery, service acceleration, and protection against various types of attacks 29
3-5 A Web application firewall sits between the end user and the application server. 29
3-6 Threat model diagram for a simple system of a Web application interface sitting in front of an Edge node. The Web API has access to a database. 34
3-7 Cyber killchain 35
3-8 HTTP Smuggling attack diagram 37

5-1 The attacking client is an HTTP client implemented in python 3. The defending server is an Azure Function instance sitting in front of an Edge node. 43
5-2 Concepts described by the grammar 47

5-3 Azure Edge WAF policy with protection against a HTTP Request Smuggling Attack disabled 50

6-1 Proof of Concept search with a population size of 4 and 10 generartions. The best solution is a request with “ExpectedOperation”. The fitness is 8.2 59
6-2 The average and standard deviation of critical parameters 60
6-3 The average and standard deviation of critical parameters 63
6-4 The average and standard deviation of critical parameters 67

A-1 Threat Model Diagram 73
A-2 Theat Model Diagram for client to edge request interaction 74
A-3 Theat Model Diagram for Web API to Database request interaction . 75
A-4 Theat Model Diagram for Edge to Web API request interaction . . . 78
A-5 Theat Model Diagram for Web API to Database response interaction 81
A-6 Theat Model Diagram for Client to Edge response interaction 84

List of Tables

5.1 Application Server API 49
5.2 internal function to emulate a cross script attack through HTTP smuggling. 49
5.3 Parameters for each experiment run. 51
5.4 The 3 phases of a killchain. In the exploit phase of the DoS attack, if the attack did not take down the service but slowed it down, the algorithm measures the impact of the attack and attempts to double down on the effort for up to 2 attempts. 53
5.5 Payoff Table. 55

6.1 Experimental Setup. The setup covers the baseline of random search, the effect of different strategies in GE and the paramater sensitivity of population and generation sizes. The size of the search space remains relatively unchanged but the different grammars have different search restrictions (See Figure 5-2 for Grammar details) 57
6.2 Application Server Action Definition 58
6.3 Actions and Results 58
6.4 Result Payoff 58
6.5 Genetic Algorithm with a cautious attack and population size 20, 20 generations 8 iterations to execute strategy - (Figure 6-2) 61
6.6 Summary of results for Denial of Service experiment 61
6.7 Distribution of operations run for an attacker with a persistant strategy and an application server vulnerable to DoS only. 65

6.8 Summaryf of Experiments: Genetic Algorithm with a cautious attack and population size 20, 40 generations 4 iterations to execute strategy - (Figure 6-3) 66
6.9 Mission results for a stealthy strategy. The application is vulnerable to all attacks - Best Solution (Figure 6-4a) 69
6.10 Mission results for a persistant strategy. The application is vulnerable to all attacks - Best Solution (Figure 6-4c) 69
6.11 Mission results for a stealthy strategy. The application is vulnerable to DoS attacks only - Best Solution (Figure 6-4b) 69
6.12 Mission results for a persistant strategy. The application is vulnerable to DoS attacks only - Best Solution (Figure 6-4d) 69

A.1 An adversary can gain unauthorized access to configure resources. . . 74
A.2 An adversary can deny actions on Cloud Gateway due to lack of auditing. 74
A.3 An adversary may spoof an system administrator and gain access to the system management portal. 74
A.4 An adversary can gain unauthorized access to database due to lack of network access protection 75
A.5 An adversary can gain unauthorized access to database due to loose authorization rules 75
A.6 An adversary can gain access to sensitive PII or HBI data in database 76
A.7 An adversary can gain access to sensitive data by performing SQL injection 76
A.8 An adversary can deny actions on database due to lack of auditing . . 76
A.9 An adversary can tamper critical database securables and deny the action 76
A.10 An adversary may leverage the lack of monitoring systems and trigger anomalous traffic to database 77
A.11 An adversary may gain unauthorized access to Web API due to poor access control checks 78

A.12 An adversary can gain access to sensitive information from an API through error messages 78
A.13 An adversary can gain access to sensitive data by sniffing traffic to Web API 79
A.14 An adversary can gain access to sensitive data stored in Web API's config files 79
A.15 Attacker can deny a malicious act on an API leading to repudiation issues 79
A.16 An adversary may spoof Cloud Edge Gateway and gain access to Web API 79
A.17 An adversary may inject malicious inputs into an API and affect downstream processes 79
A.18 An adversary can gain access to sensitive data by performing SQL injection through Web API 80
A.19 An adversary may gain unauthorized access to Web API due to poor access control checks 81
A.20 An adversary can gain access to sensitive information from an API through error messages 82
A.21 An adversary can gain access to sensitive data by sniffing traffic to Web API 82
A.22 An adversary can gain access to sensitive data stored in Web API's config files 82
A.23 Attacker can deny a malicious act on an API leading to repudiation issues 82
A.24 An adversary may spoof Database and gain access to Web API 83
A.25 An adversary may inject malicious inputs into an API and affect downstream processes 83
A.26 An adversary can gain access to sensitive data by performing SQL injection through Web API 83

A.27 An adversary may spoof an system administrator and gain access to the system management portal. 84
A.28 An adversary can gain unauthorized access to configure resources. . . 84

Chapter 1

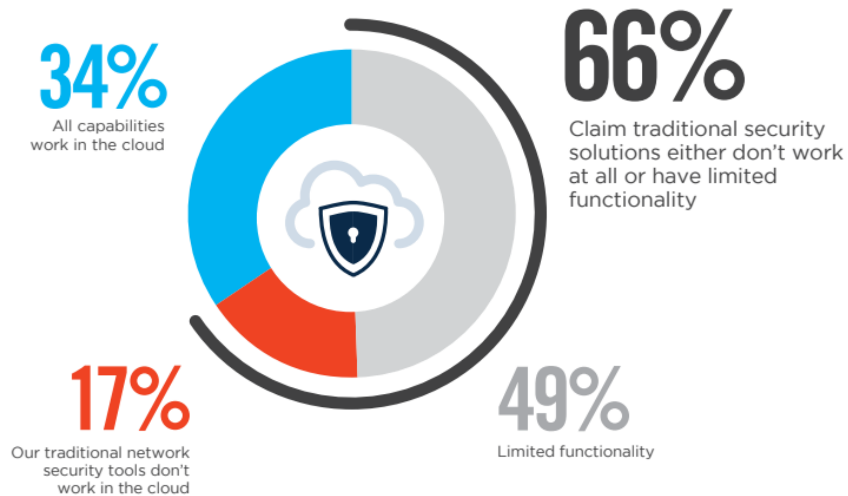
Introduction

The rise of security incidents over the years is reflected by the rapid proliferation of the cloud. As more organizations are moving their workload and data to the cloud, so rises the operational cost and growing pains of maintaing a good security and keeping up with an ever changing security landscape.

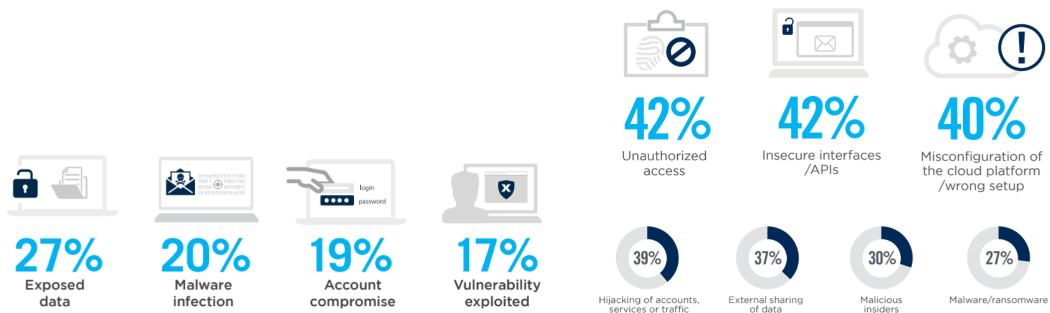
1.1 Motivation

In a report published in 2019, the Cybersecurity Insdiers found that 93% of organizations are moderately to extremely concered about cloud security [3]. Two thirds of those organizations claim that traditional security solutions either don't work or are too limited as shown in Figure 1-1a. In terms of security threats, 27% of the biggest threats relate to data exflitration. 17% of attacks exploit an implementation specific vulnerability as shown in Figure 1-1c.

While attacks are becoming more prominent, public clouds are less attractive to attack than on-premise environments, prompting organizations to move their workload to a safer haven. A Forrester research shows that only 12% of breaches target public cloud environments and that 37% of global infrastructure decision makers cite improved security as an important reason to move to the public cloud.[4].



(a) How well do your traditional network security tools/appliances work in cloud environments? [3]



(b) Biggest cloud security threats [3]

(c) Sources of vulnerabilities [3]

Figure 1-1: Cloud Security report

When it comes to securing their own workloads, organizations today are provided with a mix of tools, but often the tools are too complex and organizations don't have the expertize to use them properly. According to a Gartner report, customers of these services often do not know how to use them securely [5]. That along with a shortage of cybersecurity skills [6] highlights the need for innovative solutions to scale security as the cloud grows.

1.2 Research Goal

The goal of this thesis is to demonstrate that cloud computing backed with evolutionary algorithms trained to understand, generate, detect and prevent threats can provide greater security benefits than today's rule based and static defense solutions.

Sophisticated attacks today require intelligently targetting an endpoint, reverse engineering its application and protocols, scanning a known vulnerabilities and exploiting them. Today automation to scan known vulnerabilities exists but requries human supervision to sort through the data and generate a rule based attack or defense plan depending on which side of the threat they are sitting on [7]. Without a search gradient, weak automation will use brute force methods to find a vulnerability while a strong automation requires a lot of supervision.

1.2.1 Research Questions

The goal of this research is to demonstrate that grammatical evolution can navigate through a search space of attacks more efficiently than random search, and can beat a static defense at small scale. It is also to demonstrate that grammatical evolution can scale as we broaden the search space. This research aims to motivate the study of coevolution as a means of defense for cyber security in the cloud to complement or even replace existing static rule based defenses in Web Application Firewalls. We will assert that the same heuristics developped to create a dynamic attacking agent through grammatical evolution can be applied as a means of defense. We will leave

the demonstration for future work.

1.2.2 Research Approach

This research takes a holistic view of system security in the cloud. We expand on the different subsystems involved in a cloud application, how they are chosen by a system operator, how they interact with each other, and what they look like from an attacker's point of view. We then focus our scope to three threat scenarios chosen because of their prominance, as well as an existing gap in methods to secure workloads and data properly as shown in chapters 3 and 4.

We develop a methodology to mimic an attacker using grammatical evolution and create a web service protected by a static application firewall to run our expirements. The setup was chosen to emulate a real threat scenario at a small scale, by re-implementing known vulnerabilities and scaling down the resources required to exploit them for the purpose of the experiment as described in chapter 5. We then propse a methodology to change the the web application firewall from taking static defense to a more dynamic one by applying the grammatical evolution heurisitcs developped in this research.

1.3 Thesis Structure

Chapter 1 provides an introduction to the problem space, a motivation behind the work, and the strategy used to achieve the thesis goal. Chapter 2 presents related work in evolutionary algorithms and cybersecurity. Chapter 3 introduces the subsystems used in this research and explores them through a literature review of the technologies described. Chapter 4 explores a prominent case study for the threat scenario explored. Chapter 5 explains the methodology behind the expirements. Chapter 6 delves into the expirements and their results. Chapter 7 presents closing arguments, limitations and future work.

Chapter 2

Related Work

Censorship evasion

Geneva is a genetic algorithm used for censorsip evasion. On-path censorship is a method of censorship that sits outside the client-server communication path and monitors copies of packets to analyze and interdict them when needed [8]. Interdiction occures when the censor injects a new packet to interfere with a connection it deems inappropriate. Geneva plays a cat and mouse game with censors by using Grammatical Evolution to manipulate client-side packets to confuse them into not disrupting the communication. Heuristics used in geneva are similar to ones in this study. Geneva's approach is to operate in action trees. An action tree encapsulates the modification scheme for an incoming request. When a request matches a trigger, it enters as an action tree and is modified by the sequence in that tree. Each request triggered performs an in-order traversal of the corresponding Action Tree and the leaf node contains the final result. A strategy in Geneva is a combined forest of action tree-trigger pairs.

A cognitive approach for botnet detection using Artificial Immune System in the cloud

A botnet is a large group of computers infected with malicious software, used to perform illegal activities such as Distributed Denial of Service attacks, or information

theft. A research published in 2014 proposed a heuristic inspired by immunology for protecting the cloud against botnets[9]. The study uses Artificial Immune Systems (AIS)[10] to model immune network theory, a negative selection mechanism and clonal selection principles to create a basic immune response against the threat of botnets.

Chapter 3

Literature Review

In this chapter, we will provide a literature review of our system components, starting with the cloud infrastructure. We explore the different types of services that exist to build and run an application server. We explain Edge Computing and how system builders can improve the performance and security of their web service by having a gateway between the user and their server. We then review web application firewalls (WAF), like ones sitting at the Edge, and introduce a widely used opensource implemenation of a WAF called OWASP. We also dig into the basics of grammatical evolution and why its application is relevant to our study. Finally we dig into cyber security threats and how they are modeled and we will review the types of threats and a modeling methodology called STRIDE.

3.1 Cloud Infrastructure

The National Institute of Standards and Techology defines cloud computing as "a model for enabling convenient, on demand network access to a shared pool of configurable computing resources (e.g, networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minial management effort or service provider interaction." [11]. Currently, the cloud computing types can be either public, private (or on-premise) or a hybrid combination of both. In this section, we will dig into the different types of setups, as well as how the various computing nodes

are interconnected.

3.1.1 Web Application Servers

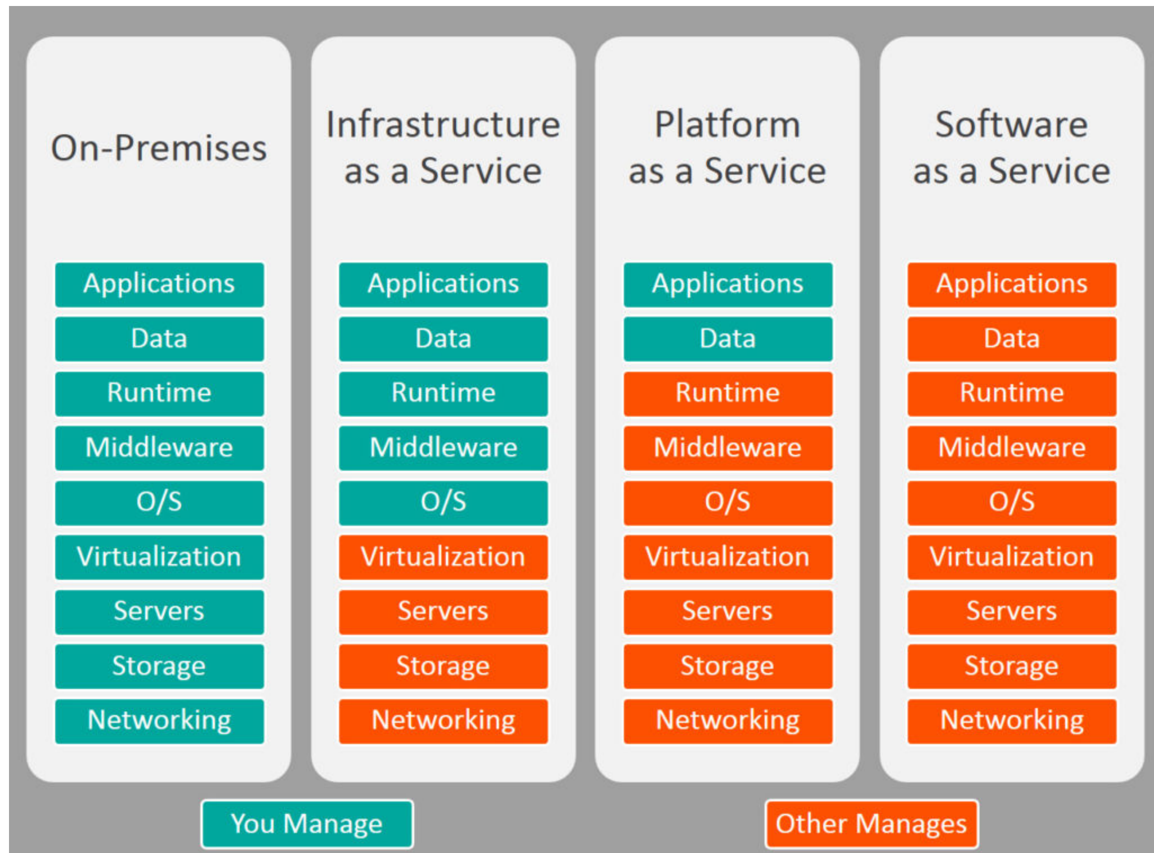


Figure 3-1: Summary of key differences of different application server infrastructures [1].

An application server is a public or private endpoint that offers a service or exhibits one or many functions over the web. Cloud providers offer three levels of abstraction, each of them allowing different levels of control. More control offers flexibilty but comes at the cost of being responsbile for the maintenance and security of the subsystem. The three types of cloud services, along with the private on-premise model, are illustrated by Figure 3-1 and described below:

1. On-Premise: The system owners manage the entire stack descibed in Figure 3-1.

2. IaaS: An Infrastructure as a Service offers the system owners complete control

over the applications and their runtime. The hardware, network, storage is leased and managed by the IaaS provider. IaaS instances are typically rented Virtual Machines (VMs) [1].

3. PaaS: A Platform as a Service offers a framework for the system owner to limit the scope of focus on the application development and data management. The system owner is only responsible for securing their application and data while the cloud provider secures the rest of the stack.

4. SaaS: A Software as a Service utlizes the internet to develop a full web application. System owners typically aim to to build their own SaaS using one the other frameworks.

The main takeaway is that on-premise applications are hosted in an infrastructure controlled by the system owner, while cloud applictions are hosted in a computing infrastructure managed by major cloud providers who have resources to build a distributed system at large scales. For an on-premise solution, the system owner requires operational expertise for all the aspects of the infrastructure. This includes security, networking and the operational challenges of dealing with a datacenter. A cloud solution takes advantage of the economies of scale the large cloud providers have access to. Major cloud providers include Amazon, Microsoft, Google, IBM and Oracle. Those big players have an uparalleled global footprint thats costs billions of dollars a year to build. Today cloud adoption continues to be one of the fastest-growing segments of system design. According to an IDC[12] research conducted in 2019:

1. By 2021, over 90% of enterprises worldwide will adpot a mix of on-premise and cloud infrastructure in their systems.

2. By 2022, 90% of web appications will be built as composite applications using public and internal API-delivered services; half of those will leverage AI and machine learning.

3. By 2022, 70% of enterprises will deploy unified VMs, Kubernetes, and multi-cloud management

4. By 2023, 10% of enterprise on-premise workloads will be supported by public cloud stacks outside of public cloud providers' datacenters and situated in customer datacenters and edge locations.

5. By 2025, 60% of enterprise IT infrastructure spending will be allocated to public cloud and a quarter of enterprise IT applications will run on public cloud services.

In the context of this research, the points highlight that as the cloud scales, workloads become more distributed, but as customers migrate to the cloud, attacks become more centralized to the handful of cloud providers. From an attacker's perspective, a centralized system generally means less potential targets to search through. For example, when it comes to deciding which target to try to overwhelm in a potential Distributed Denial of Service attack, the choice becomes more obvious. However an attacker is unlikely to have access to the resources required to overwhelm a major cloud provier who invests billions of dollars a year to guarantee their availability is close to 100% [13]. This means the next best vulnerablilty lies in the implemenation of the application server, and attack strategies will be informed by the specifics of the application and its implemenation.

Application Servers fall into two function categories. They can serve static information (videos, images, music, software updates). They can perform dynamic operations like running a full fledged application (e.g. Online banking, shopping) or provide support to the applications themselves (analytics, monitoring, etc..) In either case, those functions are typically built using RESTful APIs [14]. Attacker probing for application vulnerablilties would use those same APIs to interact with their engagement environments. The RESTful specification provides a framework for clients to run fuctions on a server but it does not provide any information on what the function does and how to call it. Tools like Swagger [15] exist to fill those gaps. Web services typically provide a public specification for their application through Swagger documentations. Like any documentation, they are prone to being incomplete and succeptible to human errors. Another way to extract calls exposed by a web server is

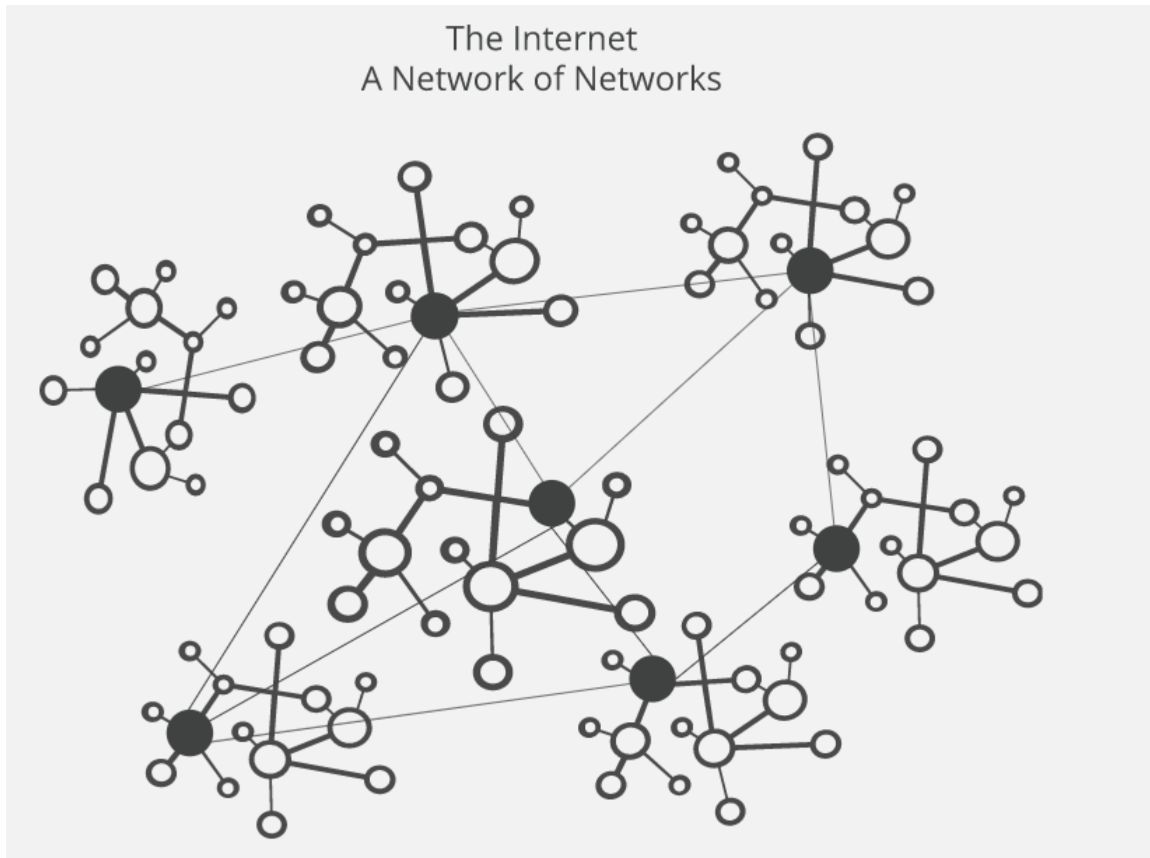


Figure 3-2: Each subnetwork is called an Autonomous System (AS) and is a large pool of routers run by a single organization, typically an Internet Service Provider [2]

with the help of profiling tools like RapidAPI [16].

3.1.2 Edge Computing and BGP Networks

The internet is comprised of many interconnected nodes that sometimes need to talk to each other. Packets need a way to naviated these nodes to go from their source and destination. This method of routing is a achieved with a protocol called the Border Gateway Protocol or BGP[2]. BGP works like the postal service, navigating through clusters of nodes called Autonmous Systems. Each of these systems is assigned a number that can be thought of as zip codes, to keep with the postal service analogy. In reality, they are typically created and controlled by Internet Service Providers. Those providers form a peer to peer network and have peering contracts with one another to create routes. Routing from one peer network to another largely

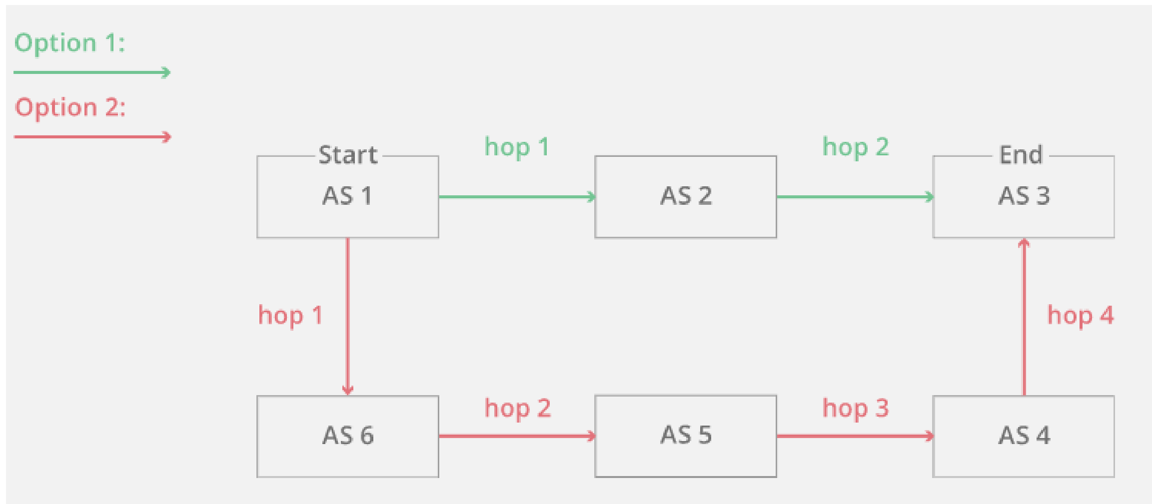


Figure 3-3: This Figure shows 6 AS. If AS1 need to go to AS3, It has 2 different options [2]

depends on performance metrics but also heavily takes into account the cost of the peering contract. A packet going from Boston to San Francisco will hop through many Autonmous Systems without necessarly following the fastest route along the way [17].

The experience for the end user can be suboptimal if an Autonmous System perfers peering to a cheaper route than a faster, more expensive one. Edge nodes are globally distibuted proxy nodes that tend to be close to the end users and aim to normalize the experience for services sitting behind them. Application Servers paired with Edge nodes benefit from performance enhacements by having users terminate their connection at the node instead of the application server if the node is closer. For example, a user in Boston looking to connect to a server located in San Fransico will benefit from having their connection terminate in an Edge node in Boston. The Edge node has a direct connection to the application server in San Francisco, therefore skipping BGP routing. The Edge node also can contain cached content that the application server wishes to deliver as close to their user as possible. Example of cached conent is music, video, images, or software updates. This is called last-mile content delivery [18].

In addition, application servers sitting behind an Edge node recieve security en-

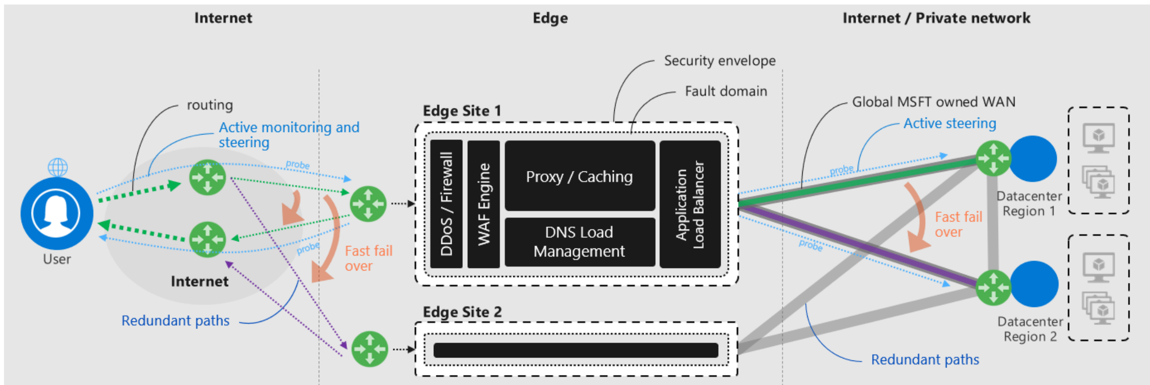


Figure 3-4: An Edge node sits between the user and the service and provides content delivery, service acceleration, and protection against various types of attacks

hancements. Not only does the distributed network act as a natural line of defense against Distributed Denial of Service attacks (see 3.4), but by virtue of being the first point of contact, Edge nodes act as gatekeepers against a would be attacker. Edge services are offered by Cloud Providers such Amazon AWS, Microsoft Azure, Google Cloud. They are also offered by specialized companies like Akamai or CloudFlare.

3.2 The Web Application Firewall

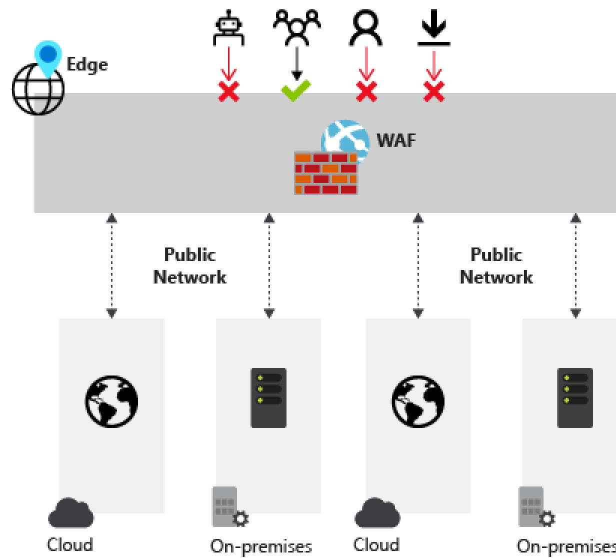


Figure 3-5: A Web application firewall sits between the end user and the application server.

A Web Application Firewall (WAF) is a software or a network device that enhances the security of web applications. Unlike a traditional firewall that typically operates on the networking layer (TCP/UDP), WAFs operate at the application layer (HTTP). As more companies move their workloads to the cloud, attackers have shifted their attention to the implementation of applications. According to a study conducted by Trustware Global Security, a cybersecurity consulting firm, over 60% of all cyber attacks target web applications [19]. In terms of location, in 2019, 54% of attacks targeted on-premise corporate environments, down 2% from 2018. In contrast, cloud services accounted for 20% of attacks, from 7% the previous year. These statistics highlight a need for cloud based web application firewalls to protect workloads with real-time monitoring, logging, debugging, and access control.

A WAF can run on the same server as the application it is protecting, or on a proxy server sitting in front of it (see Section 3.1.2). It is composed of a firewall engine as well as a managed ruleset that defines the detection rules against common attack patterns. One such ruleset is the Open Web Application Security Project Core Rule Set (OWASP CRS) [20]. ModSecurity[21] is a popular opensource WAF engine developped by the maintainers of the OWASP ruleset. Cloud providers like Amazon AWS[22], Microsoft Azure[23] and Google[24] offer their own WAF engines that run the OWASP Rule Set. For this research, we use Microsoft Azure's Web Application Firewall [23]. The advantage of using an opensource ruleset is that it is battle tested and exhaustive. There is little benefit to propriartary rulesets. Herd immunity is the heuristc to follow since the majority of attacks make it to the public domain [25].

Appendix B.1 shows an example rule that defines detection and projection against HTTP Request Smuggling. Rules from the OWASP ruleset follow the syntax below:

```
‘‘SecRule VARIABLES OPERATOR [ACTIONS]’’
```

A rule typically uses regex matching to detect if the argument of a request matches a known attack pattern. For example, to detect HTTP Request Smuggling, the rule checks if the argument of a call ends with a new line and starts with a new REST call (get,post or head, etc...). The last parameter in the rule definition above specifies

what actions should be taken if a condition is matched. The actions can be simply logging, or it could be blocking, blacklisting, redirecting the caller.

3.3 Grammatical Evolution

Grammatical Evolution is a form of evolutionary algorithm that performs an evolutionary process on variable length strings. The strings are generated by interpreting a set of production rules written in a Backus-Naur form (BNF) or extended Backus-Naur form (eBNF). The output is then evaluated using a fitness function and compared to a population of its peers, generated randomly at first. After evaluation, selection happens. The features from the fitest samples have the highest probabilty being selected to create the next generation via crossover and mutuation. Crossover is the combination of features from two individuals to create a third one. Mutuation then randomly changes a feature according to the genome, to explore new features.

The BNF grammar, developped from ALGOL 60 [26], is composed of **terminals** which are items that will be generated and **non-termials** which are items that can be expanded.
for example:

```
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The above describes the grammar of a number. If the goal is to generate the number 12345, a fitness function can be created such that 12345 would have a payoff of 5 while 12300 and 00345 would both have a payoff of 3. A genetic algorithm will ultimately converge to the best solution. Genetic Algorithms are inspired by evolutiory processes. It is only natural that they are applied in the field of genetics to model protein sequencing and the signals sent by strings of amino-acids.[27].

In cybersecurity, a notable framework called RIVALS was created to study the dynamics of coevolution with adversarial genetic programming [28]. The framework combines adversaries, engagement environments, and competitive evolution to execute coevolution through the dynamics of adversarial engagements in cyber networks.

3.4 Cyber attacks and Threat model

In this section we will expand on application level vulnerablilties. We categorize threats into three buckets as follows:

1. Social engineering attacks that aim to compromise individuals in an organization in order to gain access to their credentials. The attacks can be highly targetted, such as an attacker creating a fake social media account that mirrors their target's real and trusted acquaintance in order to gain access to the target. They can also be cast through a broader net, such as phising emails.

2. Unsophisticated denial of service attacks such as Distributed Denial of Service attacks typically operate on the network and aim to overwhelm their target. An example of such attack is a TCP SYN flood. They are prominent because they are cheap and easy to set up. At scale, their distributed and seemingly patternless nature makes them difficult to detect and defend against. Establishing a TCP session requires a three-way handshake between the parties involved. The first packet sent should have the SYN flag set which enables synchronization of the packet sequence number between the parties involved. An attack can flood a target with many connection requests from many different locations in such a way that the target cannot keep up, causing it to saturate and enventually crash.

3. Sophisticated Application level attacks operate at the application layer and aim to exploit a vulnerablilty in the system implemenation. We will expand on these types of attacks in the following sections.

3.4.1 Threat Model

Threat modeling allows system architects to identify and mitigate security issues as early as the design phase of the system lifecycle. Threat modeling tools [29], help turn system diagrams into threat vector tables to analyze potential threats.

An industry standard method used to describe and categorize the various types of threat is called the STRIDE method [30]. It categorizes threats into six types[1]:

1. Spoofing: Involves illegally accessing and then using another user's authentication information, such as username and password.

2. Tampering: Involves the malicious modification of data. Examples include unauthorized changes made to persistent data, such as that held in a database, and the alteration of data as it flows between two computers over an open network, such as the Internet.

3. Repudiation: Associated with users who deny performing an action without other parties having any way to prove otherwise - for example, a user performs an illegal operation in a system that lacks the ability to trace the prohibited operations. Non-Repudiation refers to the ability of a system to counter repudiation threats. For example, a user who purchases an item might have to sign for the item upon receipt. The vendor can then use the signed receipt as evidence that the user did receive the package.

4. Information Disclosure: Involves the exposure of information to individuals who are not supposed to have access to it—for example, the ability of users to read a file that they were not granted access to, or the ability of an intruder to read data in transit between two computers.

5. Denial Of Service: Denial of service (DoS) attacks deny service to valid users—for example, by making a Web server temporarily unavailable or unusable. Protection against certain types of DoS threats improves system availability and reliability.

6. Elevation of Priviledge: An unprivileged user gains privileged access and thereby has sufficient access to compromise or destroy the entire system. Elevation of

[1]Description taken from https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats

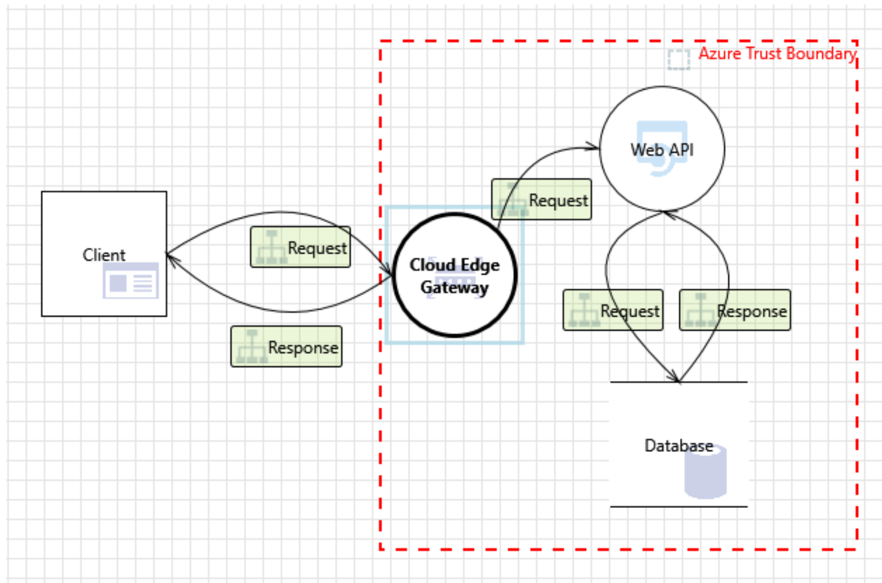


Figure 3-6: Threat model diagram for a simple system of a Web application interface sitting in front of an Edge node. The Web API has access to a database.

privilege threats include those situations in which an attacker has effectively penetrated all system defenses and becomes part of the trusted system itself.

Figure 3-6 shows an example of a threat model diagram created using the Microsoft Threat Modeling Tool. See Appendix A for details and output.

3.4.2 Advanced Perisitant Threat (ATP) and Cyber Killchain

An advanced persistant threat [31] is a prolonged network attack on a well-defined target with the intention to compromise its system and gain information about it. They are difficult to detect [32] and typically originate from highly organized and well funded efforts by governments, military organizations, and non-state actors. Because a great deal of effort and resource goes into carrying it out, attackers typically focus on high value targets with the goal of stealing assets. An attack strategy is sometimes referred to as a killchain[33] (Figure 3-7)

In Chapter 4, we examine recent case studies of such attacks. In Chapter 5 we establish a strategy by structuring our own simplified killchain.

Figure 3-7: Cyber killchain

3.4.3 Types of attacks studied in this research

Remote Code Exection

Remote code exection vulnerablilty alows attacks to run arbitrary code in a given remote application [34]. This be achieved by hijacking the logic of the web application or by exploiting a weak implemenation or vulnerablilty of underlying HTTP protocols. Remote code exceution typically present the biggest threat to a system as they give the attacker unhinged access to the entire system. The attack will have access to anything the application does, including data and other subsystems.

Information Disclosure

An information disclosure vulnerablilty exists if an attack is able to get the application to provide information it wasn't originally designed to provide [34]. This could be information about the system itself, giving the attacker an edge to exploit more vulnerablilties, or data about the organization and its customers.

Information disclosure can be achieved if the attack is able to bypass authnetication and or authorization. Many applications require the caller to supply an authentication context. This could be a user credentials (username and secret) or service credentials (service ID and secret). An attacker can obtain an authentication context by compromising the user or service or by exploiting a vulnerablilty in the authentication logic, such as weak encryption. Compromising an authentication context is not enough. The user or service requires proper authorization to a given resource for it to be comprised. An authorization bybass attack happens when a user obtains extra access to a resource they are not priviledged to have, by elevating their own priviledge for example.

Denial of Service

A denial of service attack finds and exploits a vulnerablilty that causes the application to crash or become unresponsive, denying a legitimate user access to the application. Denial of service attacks can be persistant or non-persistant [34]. A persisitant attack

disrupts a service until a repair action is taken from the services administrator. A non-persistant attack lasts as long as the attack is engaging the application with the trigger conditions that disrupt it. Denial of service attack can be achieved through CPU or Memory exhaution techniques that find and exploint algorithmic complexity of a given function's implementations.

HTTP Request Smuggling

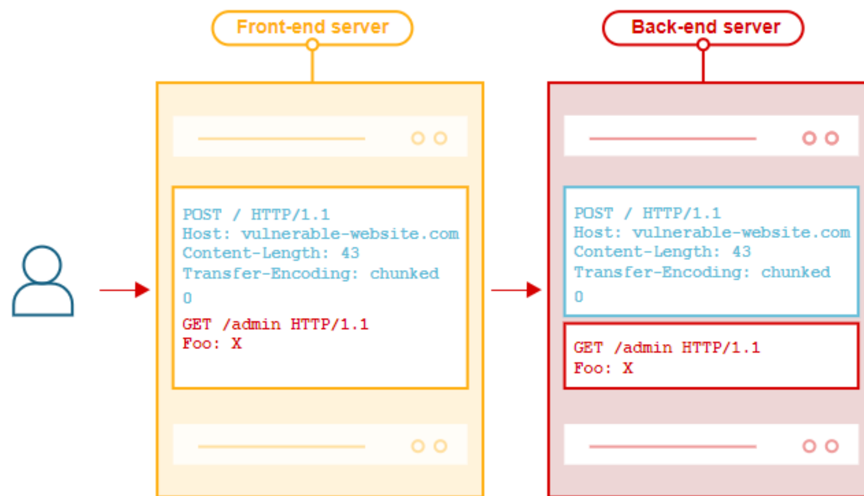


Figure 3-8: HTTP Smuggling attack diagram

HTTP attacks sit at the application layer. HTTP's RFC specifies the agreement between client and server applications consistency. One such specification is setting the HTTP packet's "content-length". This packet contains the length of the payload being sent. The RFC forbids setting this header more than once [35] but it is up to the web application implementation to respect that. In fact the vast majority of web server/proxy configurations are vulnerable to a form of HTTP smuggling [36].

HTTP Request Smuggling happens when the proxy and the server don't agree on how they read the content-length header. Figure 3-8[2] shows an example of HTTP request smuggling.

[2]taken from https://portswigger.net/web-security/request-smuggling

Below is an example of an attacker sending a packet with content-length header set twice.

```
POST http://www.target.site/somecgi.cgi HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 45

GET /~attacker/foo.html HTTP/1.1
Something: GET http://www.target.site/~victim/bar.html HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
```

From the proxy's perspective, it sees the header section of the first (POST) request, it then uses the last Content-Length header (which specifies a body length of 45 bytes) to know what body length to expect [36]

```
POST http://www.target.site/somecgi.cgi HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 45

GET /~attacker/foo.html HTTP/1.1
Something:
```

The web server sees the first request (POST), inspects its headers, uses the first Content-Length header, and interprets the first request.

```
POST http://www.target.site/somecgi.cgi HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 45
```

The body is empty. The web server answers this request, and it has one more partial request in the queue

```
GET /~attacker/foo.html HTTP/1.1
Something:
```

Since this request is incomplete (a double CR+LF has not been received, so the HTTP request header section is not yet complete), the web server remains in a wait state. The proxy now receives the web server's first response, forwards it to the attacker and proceeds to read from its TCP socket.

```
GET http://www.target.site/~victim/bar.html HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
```

From the proxy's perspective, this is the second request, and whatever the web server will respond with, will be cached by the proxy for http://www.target.site/ victim/bar.html. The proxy forwards this request to the web server. It is appended to the end of the web server's queue, which now looks as following

```
GET /~attacker/foo.html HTTP/1.1
Something: GET http://www.target.site/~victim/bar.html HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
```

The attacker succesfully completed the smuggling attack.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Case study: Equifax - An HTTP Header Attack

In March 7th 2017, Equifax was hit with the one largest security breach in history [37]. The data breach was caused by an application vulnerabilty in their customer complaint web site that ultimately allowed access to the rest of the system. 147.9 million customers were compromised, more than 40% of the population of the United States. Equifax stores highly sensitive personal data including names, addresses, dates of birth, social security numbers, and drivers licenses. 200,000 credit card numbers were also compromised.

4.1 How it happened

The attack exploited a known vulnerabilty in the customer complaint web portal. They were able to infiltrate the backend data servers through the web portal until they landed on a database containing usernames and passwords stored in plain text.

The vulnerabilty in question was in the Apache Struts framework [38]. Apache Struts is an open source development framework for creating web services. Versions of Apache Struts prior to the vulnerabiltiy patch have incorrect exception handling during file-upload attempts. An attacker can exploit the Content-Type, Content-Disposition, or Content-Length HTTP header and generate a cross-scripting attack

with a“#cmd¯” string.

4.2 Impact and Cost

147.9 million accounts were compromised, 143 million of them had their personally identifying information leaked, 200,000 had their credit card numbers leaked. In May 2019, Equifax said it spent $1.4 billion in cleanup costs after the breach [39]. This includes upgrade to the security. In July 2019, the company was required to spend an additional $1.38 billion to resolve customer claims.

4.3 Lessons learned

1. Patching vulnerabilties much be done with urgency. The attack happened weeks after the vulnerabilty was discovered and patched by Apache.

2. A system is only as secure as its weakest link. The subsystem containing the data was internal. However, compromising the web portal gave the attackers a vector of entry. The data should not have been stored in plain text just because it was in an internal server.

3. Authentifcation is not enough to protect access to data. Data governance with proper authorization is just as key. Access should only be granted on a “need to know” basis.

Chapter 5

Methodology

In this chapter we will describe the methodology used for the experiments documented in Chapter 6. Our setup takes components selected from those described in Chapter 3 and chosen to best meet our research goal defined in chapter 1.

5.1 Attacker-Defender system setup

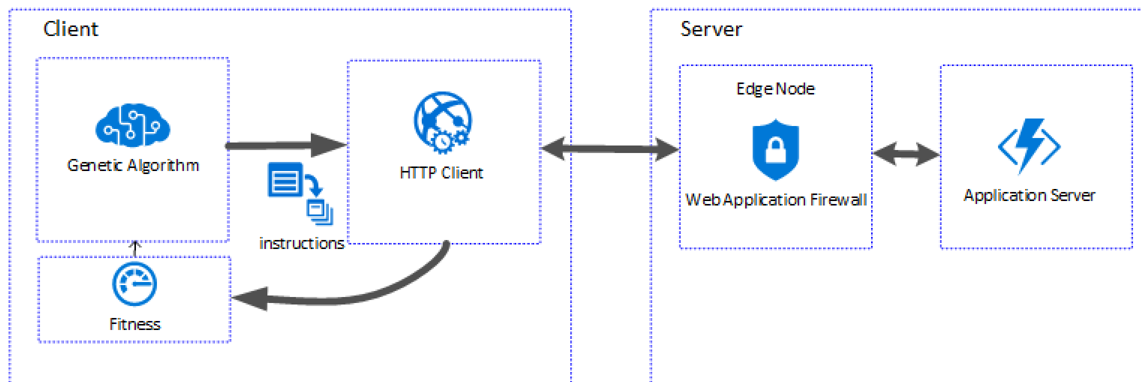


Figure 5-1: The attacking client is an HTTP client implemented in python 3. The defending server is an Azure Function instance sitting in front of an Edge node.

The setup is composed of a client that acts as the attacker, and of an HTTP Server that acts as a defender as shown in Figure 5-1. The client is driven by a genetic algorithm that generates instructions interpreted into HTTP client requests whose parameters and payload are are the subject of the evolution. The requests

first hit an Edge server running a Web Application Firewall (WAF). They are then forwarded to the web application server that then processes the request and generates a response. That response is sent back to the HTTP client via the Edge server. The client parses the response, runs it through a fitness function, feeds the results back to the genetic algorithm, and a new generation of requests is formed.

5.1.1 The Attacker: A client running a grammatical evolution algorithm

Application Runtime

At the core of the client is the genetic algorithm (GA) running a Grammatical Evolution (GE) implemented using the Donkey GE framework [40]. Donkey GE is a basic implementation of Grammatical Evolution using python3. The algorithm generates instructions based on a grammar formulated in BNF (Chapter 3.3). The instructions are interpreted as HTTP requests. The responses of those requests are evaluated through a fitness function that evaluates the returned content (payload) and meta-data (response time, status code). The payoffs are then fed back to the GE that will then create a new population for generation.

Grammar Definition

The grammar is written in BNF. It expresses how the HTTP requests sent by the attacker to the server are formed. The goal is to have the grammar run through a python engine that interprets it into HTTP requests. Throughout the course of the experiments, we evolve the grammar to express more and more complex concepts that revolve around attack missions. Our goal is for the grammar to be interpretable by both the attacking and defending algorithms. For the purpose of this research we will limit the scope of the GE application to the attacking side of the adverserial engagement. Applying this method to the defending side and studying coevolution is reserved for future work.

Throughout our experiments, we evolve the grammar four times (Figure 5-2).

With each version the grammar's expression converges from the description of a generic HTTP Request with unrestricted parameters to a complete attack strategy with parameters scoped to a mission objective. The grammar evolves as described below:

1. Generic Request: The first version of the grammar describes a generic HTTP request. A request is parameterized by its action. Another way of thinking about a generic HTTP request is with the following sentence: "Given an endpoint, make an HTTP request to a function called $\langle name \rangle$" where $\langle name \rangle$ can be any string. An unbound string search will very likely generate non-sensical function names and an endpoint should return an HTTP response with status code 404 "not found". This grammar serves to lay the groundwork for more complex expressions of attacks.

2. Specialized Request: The second version narrows the unbound action space of an HTTP request by specializing it into a REST call. This can be thought of as saying "Given an endpoint, and a set of know function names, make an HTTP request to function called $\langle name \rangle$" where $\langle name \rangle$ is a known function.

3. REST Call: The third version expands the concept and generalizes the REST call by introducing the API Action. This version also introduces the notion of "missions". Missions will inform fitness and strategy for a given API action. This can be thought of as meaning "Given an endpoint, make an HTTP request to call a function $\langle name \rangle$ that takes arguments $\langle args \rangle$ with the objective to fulfill mission $\langle mission \rangle$", where $\langle name \rangle$ is a REST call that takes arguments $\langle args \rangle$ and mission is a type of attack.

4. Missions: The final version sets up the groundwork for future work and interoperability with a GE algorithm on the defense side to create a framework for coevolution. This version rescopes the grammar away from "requests" to define "missions". The sentences can be interpreted as meaning "Given an endpoint, run a set of $\langle missions \rangle$ with their method $\langle method \rangle$ and API Action $\langle action \rangle$."

This sentence becomes very much tailored towards the specificity of attacks and how to execute them. For example, denial of service would have methods to exploit poor runtime performances, while "infiltrate" will use methods like HTTP Request Smuggling as a vehicle for a cross-scripting attack. The scope of attacks and methods can easily be expanded with this grammar.

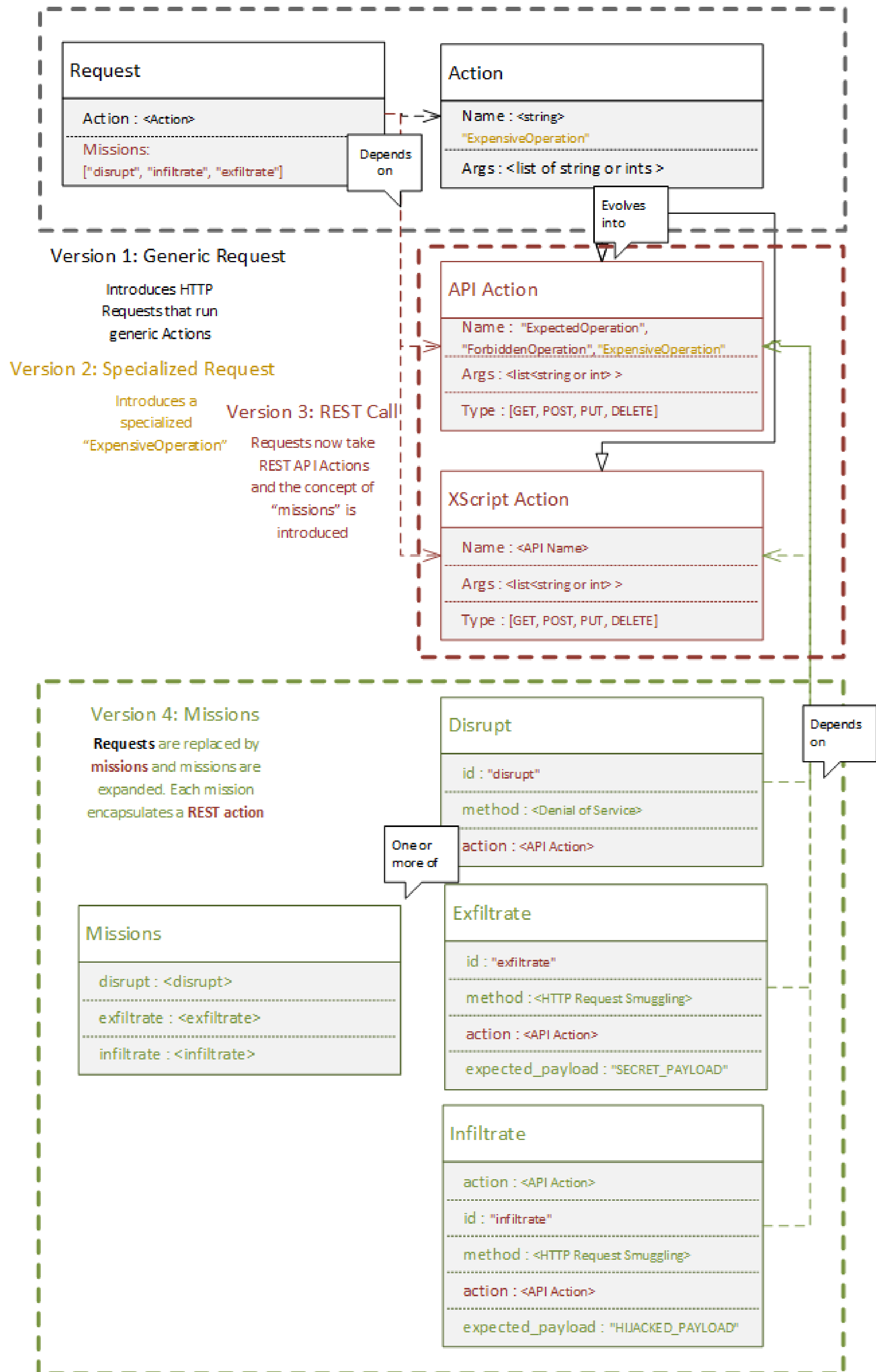


Figure 5-2: Concepts described by the grammar

5.1.2 The Defender: An application server behind an Edge node running a Web Application Firewall

The Application Server

The application server is implemented as a Platform as a Service (PaaS - see Chapter 3.1.1). Specifcally, we run an instance of an of Azure Function. Azure functions are a type of serverless computing[41] that abstracts away any underlying infrastructure and lets system owners focus solely on designing and implementing their application. The infrastructure's implementation and security are managed by the cloud provider (Azure in this case), leaving us to focus on developping and securing the application and its data. The Azure Function implements three public functions and one internal function as described in Table 5.1

The application server implements an additional placeholder function that facilitates a cross-script attack through HTTP Request Smuggling (See Chapter 3.4). Recall that a cross script attack through HTTP smuggling occurs when a proxy (Edge) server is tricked into running a hidden command on an application server (See Chapter 3.4). We emulate this flow by having the Edge server recognize an HTTP Request Smuggling attack and setting a flag in the Web Application Server to execute the hidden function when the condition occurs. The hidden function is otherwise uncallable.

Table 5.1: Application Server API

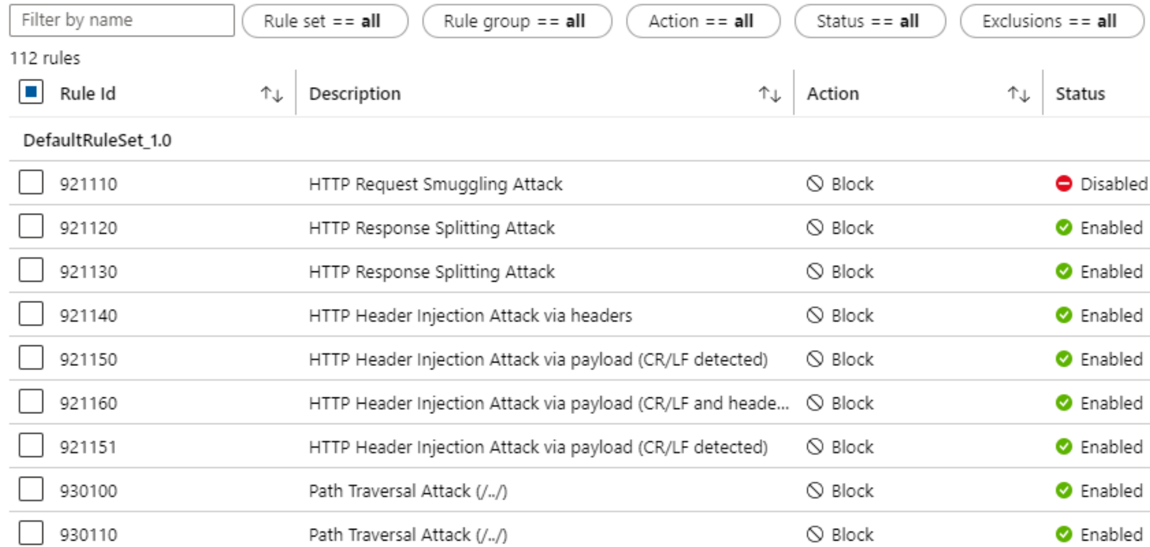
Method	**Arguments**	**access policy**	**description**
ExpectedOperation	None	public	Always returns HTTP status code 200 OK
ExpensiveOperation	string	public	The function has a high algorithmic complexity. The runtime duration is t in ms, defined as $t = 100x + 10y^2$ where x is the string length and y is the number of non alphanumerical characters.
ForbiddenOperation	None	private	Only the Edge server is allowed to call it. Returns HTTP status code 403 FORBIDDEN for all other callers

Table 5.2: internal function to emulate a cross script attack through HTTP smuggling.

Method	**Arguments**	**access policy**	**description**
UnknownOperation	None	hidden	Only executable if a cross script attack through HTTP smuggling is succesful. Application Server returns a HTTP status code 404 NOT FOUND otherwise.

The Edge Server and Web Application Firewall

The Edge server is an instance of an Azure Edge site that sits in front of the application server and provides protection against attacks through a WAF (See Chapter 3.2). For our experiment we disable protection against a HTTP Request Smuggling Attack (See Figure 5-3) to measure GEs convergence towards the attack.



Filter by name | Rule set == all | Rule group == all | Action == all | Status == all | Exclusions == all

112 rules

Rule Id	Description	Action	Status
DefaultRuleSet_1.0			
921110	HTTP Request Smuggling Attack	Block	Disabled
921120	HTTP Response Splitting Attack	Block	Enabled
921130	HTTP Response Splitting Attack	Block	Enabled
921140	HTTP Header Injection Attack via headers	Block	Enabled
921150	HTTP Header Injection Attack via payload (CR/LF detected)	Block	Enabled
921160	HTTP Header Injection Attack via payload (CR/LF and heade...	Block	Enabled
921151	HTTP Header Injection Attack via payload (CR/LF detected)	Block	Enabled
930100	Path Traversal Attack (/../)	Block	Enabled
930110	Path Traversal Attack (/../)	Block	Enabled

Figure 5-3: Azure Edge WAF policy with protection against a HTTP Request Smuggling Attack disabled

5.2 Methodology Limitations

Unless otherwise specified, the evolution runs with a population size of 20 and iterates through 40 generations (see section 5.3). The strategy also implements four phases of a cyber killchain. Each phase translates to an HTTP Request (see section 5.4). This results in a total number of requests run per experiments of 1600. A normal request takes about half a second to complete, and a disruptive request takes 10 seconds to complete since the client has to wait for the server to respond. This means a single experiment takes about 40 minutes to an hour to run and makes repeatability at scale computationally intractable with normal computing resources. We will discuss strategies to improve this constraint in future work with parallalization of independent

requests (See Chapter 7.2).

5.3 Parameters tested in Grammatical Evolution

Our grammar forms syntactically correct queries, therefore do not employ any special crossover or mutation operators. Individuals are randomly generated queries with arguments of variable length. An unconstrained search is performed on these strings due to the genotype-to-phenotype mapping process that generates syntactically correct individuals.

The parameters at play in our experiments are the population size, the number of generations, the mutuation rate, the tournament size, and the crossover probabilty. We will only vary the population size and the number of generations and leavethe rest of the parameters constant. Our experimental setup is described in Table 5.3. Our goal is to demonstrate that GE algorithms perform better than random search. As we broaden our search space by increasing the complexity of our grammar, GE can scale where random search cannot. For this purpose, setting the crossover, mutation probabily, tournament and elite sizes to their standard values is sufficient. Having the GE algorithm stuck in local optima does not counter the research goal of proving its effectivess.

Table 5.3: Parameters for each experiment run.

	Denial Of Service			**All Attacks**			
Experiment	GE without strategy	GE with strategy	GE parameter sensitivity test	stealthy strategy vs all vulnerabilties	stealthy vs DoS only	persistant strategy vs all vulnerabilties	persistant strategy DoS only
population size	4		20	20			
generations	20		4	40			
max length	10			10			
elite size	10			10			
tournament size	2			2			
crossover probability	0.8			0.8			
mutation probability	0.1			0.1			

5.4 Attack selection and Strategy

Our experiments are centered around the Denial of Service attack since it can be achieved through CPU or Memory exhaustion techniques that find and exploint algorithmic complexity of a function's implementations. Disruption gives us a direct measure of fitness via the response time for the request. The GE algorithm's search gradient can follow the resulting latency from a request it generates. The addition of infiltration and exfiltration via HTTP Request Smuggling broadens our search scope and serves to demonstrate that GE scales as we increase the number of attacks described by the grammar. The attack selection also puts into play the high profile attack described in the case study see chapter 4). For our strategy, we implement a killchain scaled down to the the core phases below (see chapter 3.4):

1. Phase 1 - Reconnaissance: The first query is sent to act as a baseline for reconnaissance. The results will inform the subsequent phases.

2. Phase 2 - Attack: In this phase, the attacker transforms the query to make it harmful to the server.

3. Phase 3 - Exploit: In this phase the query doubles down on an attack if it was not succesful in the previous phase. For denial of service, this means re-applying the disruptive transformation. In this case, we double the argument size. Phase 3 is attempted at most twice. If an attack is not succesful at taking down an endpoint but is able to slow it down, the attacker is able to measure the impact of the first query and of any transformation applied.

The strategy executes the killchain with at most four HTTP request calls, as shown in the table 5.4.

Table 5.4: The 3 phases of a killchain. In the exploit phase of the DoS attack, if the attack did not take down the service but slowed it down, the algorithm measures the impact of the attack and attempts to double down on the effort for up to 2 attempts.

Request #	Denial of Service	Infiltration	Exfiltration
1	Reconnaissance		
2	Attack		
3	Exploit		
4	Exploit		

5.5 Search space

Our goal is to demonstrate that Grammatical Evolution can navigate through a broad attack search space better than random search. We start with a narrow searchspace and broaden it as we expand our grammar (see section 5.1.1)

The proof of concept explores three HTTP Requests, of which only one will succeed. The size of the search space is therefore $n_{requests} = 3$. Next, we narrow the HTTP Requests to a single action known to have a high algorithmic complexity. The request search space is reduced to a size of $n_{requests} = 1$. The call however has an argument of type "string" of variable length of up to 10 characters. A string is defined as a combination of uppercase letters with $n_{upper} = 26$ possible choices, lowercase letters with $n_{lower} = 26$, digits with $n_{digits} = 10$ possible choices, and 21 special characters with $n_{special} = 21$ possible choices. A 10 character string creates a search space of $n_{string} = (n_{upper} + n_{lower} + n_{digits} + n_{special})^{10} \approx 10^{19}$ possible values.

The next experiment further expands the request search space by covering all $n_{api} = 3$ functions supported by the application server as well as the $n_{xscript} = 1$ function the attacker aims to infiltrate. The total number of functions is $n_{requests} = n_{api} + n_{xscript} = 4$. We also introduce the concept of missions. The $n_{missions} = 3$ types of missions are "infiltrate", "exfiltrate", "disrupt". Our search space size becomes: $n = n_{requests}.n_{missions}.n_{requests} \approx 10^{20}$

Finally, for our last experiment, we rescope the grammar around the mission and away from requests. The number of missions and API calls remains unchanged and the order of magnitude of the search space size remains $n \approx 10^{20}$.

5.6 Measure of fitness

We set the payoffs for metrics that give off strong signals of success to have an order of magnitude of 10^3. Penalities for metrics that give off strong signals of failure have an order of magnitude of 10^2. Payoffs/penalities for metrics that give off a weak signal of success/failure have an order of magnitude of 10^1. The scores were assigned based on desirabilitly of outcome. For this study we consider all attacks to be equally weighted. A strong measure of success is succesfuly executing an attempted attack. We reward this condition with highest payoff (e.g. getting a HTTP status code response of "500 - Server Error" for denial of service attack). A strong measure of failure is defined if the objective of the attack was not achieved. Here there is no intrinsic consequence to failing. It is mearely undesirable. A weak meaure of success is taking a step in the right direction. For Denial of Service, that means creating latency. For each milisecond of latency we award a small payoff to create a search gradient. A weak measure of failure is taking a step in the wrong direction and generating noisy requests. A noisy request is a request that doesn't accomplish any objective and does not give any meaningful data. An example of a noisy request is one that results in the server returning a response with HTTP status code of "404 - Not found". Assigning a negative payoff for these requests should filter them out with each generation. Given the payoffs above, we defined fitness to be the sum of payoffs of the signals described in Table 5.5.

The type of startegy applied will also inform the fitness. A cautious strategy penalizes failure and wasted iterations. The attacker's goal is to remain undetected. A bold strategy gives no penalties. The attacker's goal is to disrupt as quickly as possible. The fitness of types of strategies will be measured and compared. In the first experiment we compare an evolutionary algorithm that has strategy with one that doesn't 3.4. The bold and cautious strategies model persistant and stealthy attacks [42]. The payoff equations associated with each strategy is represented in Table 5.5

Table 5.5: Payoff Table.

	Strategy	
Condtion	**Bold**	**Cautious**
An attempted attack is succesful	1000	1000
An attempted attack failed	0	-100
The request exposes a vulnerabilty other than the attempted attack.	100	100
Request failed	0	-10

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Experiments

Table 6.1: Experimental Setup. The setup covers the baseline of random search, the effect of different strategies in GE and the paramater sensitivity of population and generation sizes. The size of the search space remains relatively unchanged but the different grammars have different search restrictions (See Figure 5-2 for Grammar details)

	PoC	**Denial Of Service**				**All Attacks**				**Mission Centric**			
Section	**6.1**	**6.2.1**	**6.2.2**	**6.2.2**	**6.2.2**	**6.3.1**	**6.3.1**	**6.3.2**	**6.3.2**	**6.4**	**6.4**	**6.4**	**6.4**
Grammar	generic request	N/A	special request			rest call				mission			
population	4	N/A	4	4	20	20	20	20	20	20	20	20	20
generations	10	N/A	20	20	4	40	40	40	40	40	40	40	40
strategy	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
server vulnerabilities	All	All	All	All	All	All	DoS	All	DoS	All	DoS	All	DoS
search space	4	10^{19}	10^{19}	10^{19}	10^{19}	10^{20}	10^{20}	10^{20}	10^{20}	10^{20}	10^{20}	10^{20}	10^{20}

In this chapter we dive into the experiment results using the methodology described in Chapter 5. We start with a simple HTTP request and evolve it into a full fledged set of attack "missions" as described by Figure 5-2.

Table 6.1 highlights the experimental setup. Section 6.1 is a proof of concept implementing a generic HTTP client driven by GE. In Sections 6.2.1-6.2.2, we build on the proof of concept to investigate the Denial of Service attack. We contrast and compare random search with our GE algorithm with and without a strategy. Finally we test parameter sensitivity by increasing the population size and lowering the number of generations, keeping the total number of requests generated constant. Sections 6.3.1-6.3.2 broadens the breadth of attacks by introducing HTTP Request

smuggling as a means of generating a cross-script attack (infiltration), and an information disclosure attack (exfiltration). We contrast and compare two strategies: A "cautious" strategy that aims to be stealthy, and a "bold strategy" that aims to be disruptive. The bold and cautious strategies model persistant and stealthy attacks [42]. Finally, in Section 6.4 we run the experiments with a modified grammar scoped around formulating an attack mission statement rather than an HTTP Request (see Figure 5-2 for details).

6.1 Proof of Concept

The proof of concept serves to set up the first building block for our grammatical evolution. The grammar describes a simple request defined in the "Version 1" section of Figure 5-2. The options for "Action" are chosen from a set of pre-selected strings. The application server will return 200 OK to only one of the strings. The experiment's objective for GE is to return an instruction that generates a request with a successful response. Table 6.3 shows the actions the GE can choose from along with the expected response from the application server. Table 6.4 shows the reward system for this setup.

Table 6.2: Application Server Action Definition

Table 6.3: Actions and Results

Action		
Name	**Args**	**Status Code**
ExpectedOperation	[ ]	200
UnkownOperation	[ ]	404
ForbiddenOperation	[ ]	403

Table 6.4: Result Payoff

Name (Code)	Value
OK (200)	10
Other	-1

With a simple grammar and a simple fitness function GE can return a desired HTTP request. Figure 6-1 shows a successful outcome after 1 generation, as expected given that the $population_size > search_space$. The next experiment introduces a function that takes arguments. The function has high algorithmic complexity and makes the application server vulnerable to DoS attacks. The experiment also introduces the notion of strategy to the GE algorithm (see chapter 5.4).

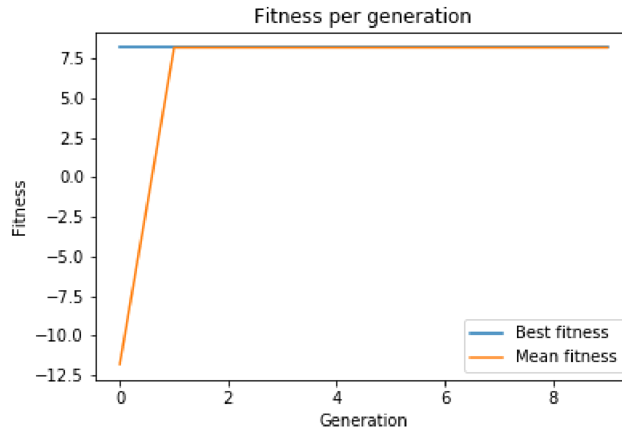
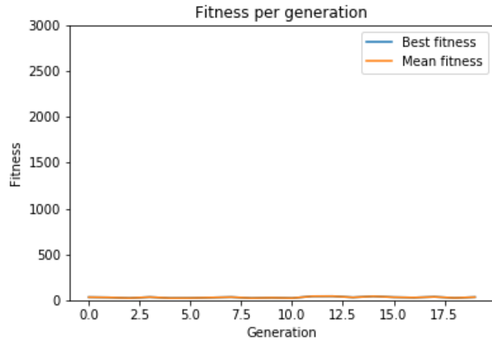


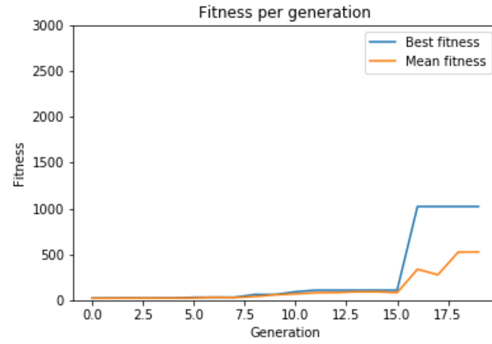
Figure 6-1: Proof of Concept search with a population size of 4 and 10 generartions. The best solution is a request with "ExpectedOperation". The fitness is 8.2

6.2 Denial of Service: Exploiting an expensive operation

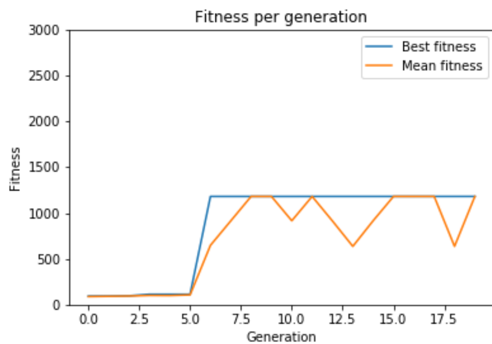
In this Experiment, the client runs a function in the application server that was designed with a high algorithmic complexity. The function has runtime duraction $t(ms) = 100x + 10y^2$ where x is the length of the string argument and y is the number of non alphanumerical characters in the string. The name of the operation is "ExpensiveOperation" and is described in Chapter 5.1.2. We set the HTTP Request's Action to "ExpensiveOperation" and generate string arguments of length between 1 to 10. A string is defined as a combination of $n_{uppper} = 26$ uppercase letters, $n_{lower} = 26$ lowercase letters, $n_{digits} = 10$ and $n_{special} = 21$ An argument can have $n = (n_{uppper} + n_{lower} + n_{digits} + n_{special})^{10} = 15 \times 10^{18}$ possible values (see chapter 5.5). If an operation takes more than 10s to execute, the server crashes and the denial of service attack is executed successfully. It would take a string of exactly 10 special characters to crash the service.



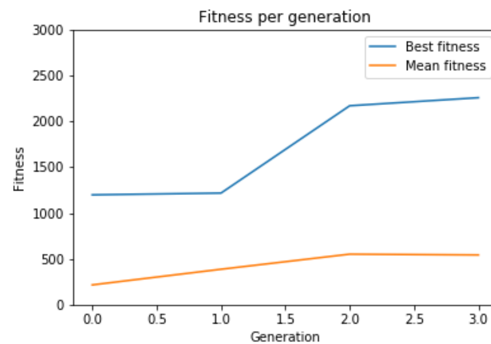
(a) Random search



(b) GE with no strategy



(c) GE with strategy



(d) Population size of 20 and 4 generations

Figure 6-2: Denial of Service attack with Randon Search and Grammatical Evolution. (a) Random search has a mean fitness of 32.715. The strings are generated uniformly at random. (b) GE with a population size of 4 and 20 generations. No strategy is applied. (c) GE with a population size of 4 and 20 generations. A strategy of doubling the argument size if a request is significantly slower than another is applied. (d) GE with a population size of 20 and 4 generations. Strategy from (c) is applied

6.2.1 Random Search

Disruption happens if the string argument has 10 special characters. A random search has a $\frac{n_{special}^{10}}{n} = \frac{21^{10}}{(15 \times 10^{18})} = 0.0001\%$ chance to generate a string with 10 special characters. Figure 6-2a shows a low flat curve for random search. The mean fitness is 32.

6.2.2 Grammatical Evolution

We first run grammatical evolution as is, with population size of 4 and iterating through 20 generations. The best individual is a request with argument "#%_!_{))}"

Table 6.5: Genetic Algorithm with a cautious attack and population size 20, 20 generations 8 iterations to execute strategy - (Figure 6-2)

Parameters			**Request**		**Results**	
Strategy	**Pop**	**gen**	**Name**	**Args**	**tries**	**Fitnesss**
No	4	20	“ExpensiveOperation”	[“#%_!_{))}”]	64	1021.78
Yes	4	20	“ExpensiveOperation”	[“))”]	24	1180.60
Yes	20	4	“ExpensiveOperation”	[“{!$+=”]	40	2255.36

and is created after 16 generations, or 64 total requests. Applying a strategy to double the argument size to a request that is significantly slower than expected hastens the process and a champion is generated after 6 generations, or 24 total requests. The champion has argument “))” that would have needed to be multiplied 3 times to generate a string with at least 10 special characters. If we increase the population size from 4 to 20 and reduce the number of generations from 20 to 4, the GE has an even greater performance. It is able to generate an individual that will disrupt the service with less transformations, resulting in a fitness of 2255 - twice as high as a GE with lower population size. The champion is a request with argument “{!$+=”. Performing 1 transformation to this request is enough to disrupt the service. A higher population gives the GE more room to explore the broad search space of strings to the kind that will disrupt the server. A strategy exploits what makes the string disruptive and amplifies the effect of disruption. Table 6.6 summarizes how exploiting a DoS Vulnerability is better achieved with a GE than with random search. Furthermore, a GE with a strategy and a big enough population to explore provides an even greater outcome.

Table 6.6: Summary of results for Denial of Service experiment

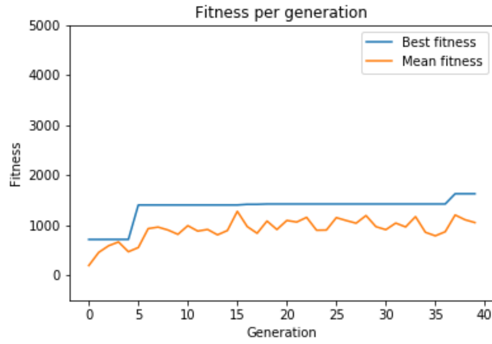
Method			**Mission**		
Name	**pop**	**gen**	**Attacks**	**Best Fitness**	**Note**
Random Search	N/A	N/A	“ExpensiveOperation”	37	
GE with no strategy	4	20	“ExpensiveOperation”	1021.78	After 64 requests
GE with strategy	4	20	“ExpensiveOperation”	1180.60	After 24 requests
GE with strategy	20	4	“ExpensiveOperation”	2255.36	After 40 requests

6.3 Increasing the breadth of attacks and strategies

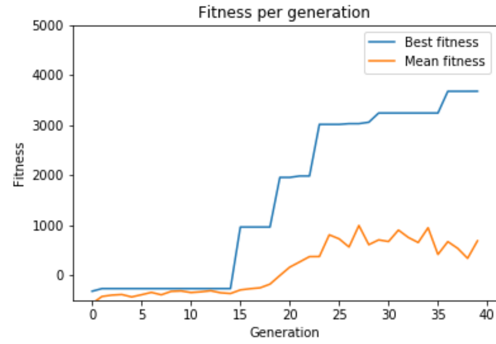
Up until now, the attacker was given a vulnerable endpoint and the goal was to exploit it by finding the argument that causes the service to break. In this experiment, we expand the scope of what an attack is to include HTTP Request Smuggling (See 3.4.3. HTTP Request Smuggling will allow the attacker to attempt cross-scripting through infiltration and information discolusre attacks through exfiltration. During infiltration the attacker will attempt to hijack the server and run their workload, a success condition is if the returned response contains the predefined payload : "HIJACKED_PAYLOAD". Exfiltration follows the same heuristic but with a different approach. If the response to a request contains a HTTP Status Code of "403 - ForbiddenOperation" during the reconnaissance phase of the killchain (see chapter 5.4 for details), the attacker will attempt HTTP Request Smuggling to gain an elevation of priviledge by having the Edge server call the restricted method on behalf of the attacker. An exfiltration attack is successful if the application server response contains the predefined payload : "SECRET_PAYLOAD".

The grammar used for this experiment builds on the previous version of the grammar that defined Denial of Service attack and introduces the attacks above. It also introduces the concept of "missions". The missions parameter carries information on the attacker's goal. Fitness function uses the server's response to determine if a goal was achieved. Actions in the new grammar encapsulate the notion of a REST API 5.1.2 as well as the definition of a script used for a cross-script attack. The mission types will infrom the strategy and has no bearing on the requests generated by the grammar. The GE algorithm will seed the requests and "strategy" will execute the killchain described in chapter 5.4.

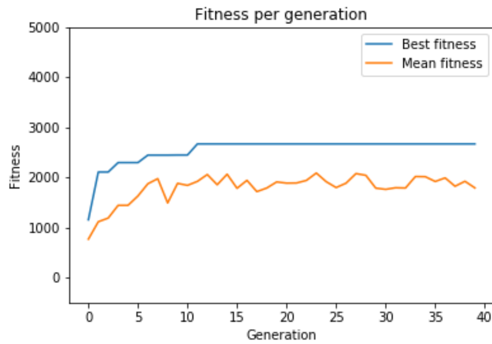
This experiment also explores two payoff models. The first payoff structure models a stealthy attacker. It favors remaining undetected by penalizing failed attacks and rewarding finding disuptive attacks as quickly as possibe. The second payoff structure models a bold strategy that favors persistance in an attacker and no penalty is



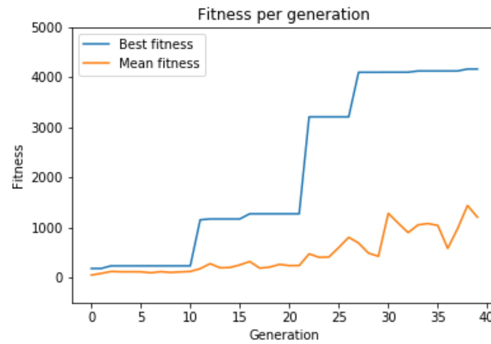
(a) Stealthy strategy on an application server with all vulnerabilities



(b) Stealthy strategy on an application server vulnerable to DoS only



(c) Persistant strategy on an application server with all vulnerabilities



(d) Persistant strategy on an application server vulnerable to DoS only

Figure 6-3: Genetic algorithm with a population size of 20 and a generation size of 40. The algorithm searches through HTTP Request smuggling and DoS vulnerabilities in a given application server endpoint. The search space has size of $n = 1.6 \times 10^{21}$

assigned to failures. This strategy can be seen as disadvantegous against a defense strategy that blacklists suspicious traffic.

6.3.1 The stealthy attacker strategy

In the first experiment of this setup (Figure 6-3a), the engagement environment is vulnerable to all attacks. In the 1st generation, the algorithm discovered the exfiltrate attack for a fitness of 716. The fitness assigns a score of 1000 for having successfully run the attack, however since only 1 request in the killchain accomplished the attack, the rest of the requests suffer a penalty. In the 6th generation, the algorithm discovers that it can both infiltrate and exfiltrate the target for a payoff of 1403. In the 36th

generatiom, the algorithm discovers it an achieve a better outcome by running an infiltration and disruption campaign for a fitness of 1627.74. This is because the fitness function rewards creating latency so although the killchain is the same for both [“infiltrate”, “exfiltrate”] and [“infiltrate, “disrupt”] campaign, the requests in the killchain that aim to disrupt the DoS vulnerability create high latency on the server. The exfiltration requests don’t have a comparable effect. Since we chose to assign the same weight for all attacks, the algorithm will favor the attack that creates latency along the way. Denial of service is harder to achieve than infiltration and exfiltration in our engagement environment. It took 36 generations for the algorithm to successful disrupt the server.

In the second experiment using this setup (Figure 6-3b), the engagement environment is only vulnerable to the DoS attack. Until the 16th generation the fitness is negative as the algorithm struggles to find an attack. In the 16th generation, the algorithm finally creates an individual that has a mission to disrupt, with an “ExpensiveOperation” and an argument with 2 special characters. Running this request through the killchain will lead to the creation of an attack that takes 3 transformations on the string argument to succeed. Recall that each transformation just doubles the string length. The algorithm progressively creates strings with more and more special characters, requiring less transformations to succesfully disrupt the server. In the 37th generation, the algorithm creates an request with fitness 3674.10. It has 8 special characters and requires no additional transformation to disrupt the server.

6.3.2 The persistant attacker strategy

In the first experiment of this setup (Figure 6-3c), in the 1st generation, the engagement environment discovers the infiltrate attack for a fitness of 1155.18. It does so by attempting all missions and is not penalized for failing to disrupt or exfiltrate, even though the missions were part of the campaign. In the 2nd generation, the approach to try all missions is kept in the algorithm’s DNA and so it discovers it can also trigger a sussful attack by running a campaign of [“disrupt”, “exfiltrate”, “infiltrate”] by running the “ForbiddenOperation” function. It succesfully completes 2 out of the

the 3 missions and is rewarded 2106.06 points. The algorithm is not able to make any meaningul gain from there since with the current reward system, looking for the Denial of Service attack is like finding a needle in haystack and all the hay is made of gold. The agorithm is not incentivized to explore more since it doesn't get much gain from it, nor does it get penalized for making requests that don't amount to anything.

In the second experiment using this setup (Figure 6-3d), the engagement environment is only vulnerable to the DoS attack. Until the 11th generation, the fitness is close to 0 as the algorithm struggles to run a succesful campaign. However, we observe in 5th generation, the algorithm is starting to discover the individuals with ExpensiveOperations in its population. It's not until the 12th generation that the GE starts to focus on Denial of Service of attack. In the 27th generation, it creates an request capable of disrupting the service without any additional transformation.

Table 6.7: Distribution of operations run for an attacker with a persistant strategy and an application server vulnerable to DoS only.

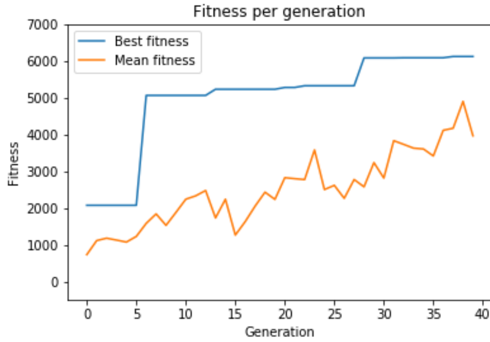
Generation	**1**	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**
ExpensiveOperation Count	5	10	6	6	9	13	18	17	16
ExpectedOperation Count	2	3	4	0	1	2	1	2	2
ForbiddenOperation Count	2	2	5	0	2	0	0	0	0
UnknownOperation Count	11	5	5	14	8	5	1	1	2

Table 6.8: Summaryf of Experiments: Genetic Algorithm with a cautious attack and population size 20, 40 generations 4 iterations to execute strategy - (Figure 6-3)

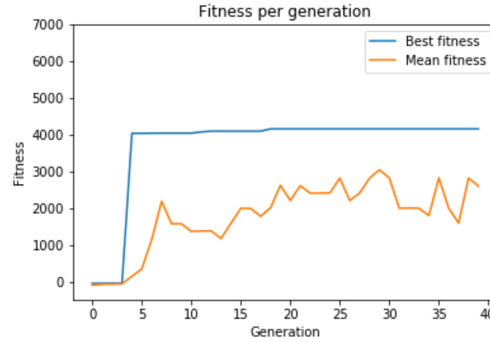
Parameters		**Mission**			**Results**
Vulnerability	**strategy**	**Name**	**Action**	**Args**	**Fitness**
all	stealthy	[“disrupt”, “infiltrate”]	“ExpensiveOperation”	2 special characters	1627.74
DoS only	stealthy	[“disrupt”]	“ExpensiveOperation”	8 special characters	3674.10
all	persistant	[“exfiltrate”, “infiltrate”]	“ForbiddenOperation”	None	2666.30
DoS only	persistant	[“disrupt”,“exfiltrate”, “infiltrate”]	“ExpensiveOperation”	9 special characters	4159.10

6.4 Rescoping the grammar from HTTP Requests to attack missions

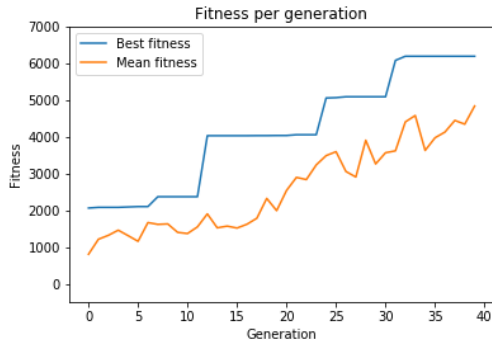
Up until this point, the GE generated a population of requests with actions and mission objectives. In this experiment, the GE generates mission contexts. Requests now take a supporting role for the mission. The context of the grammar is built around the missions objectives.



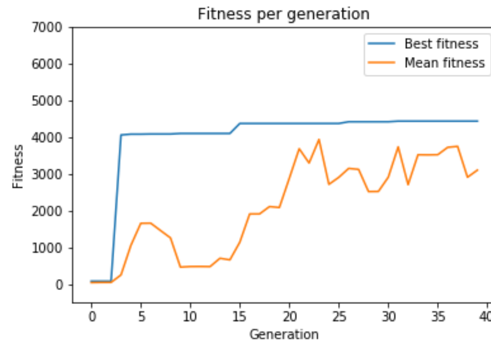
(a) Cautious strategy on an application server with all vulnerabilities



(b) Cautious strategy on an application server vulnerable to DoS only



(c) Bold strategy on an application server with all vulnerabilities



(d) Bold strategy on an application server vulnerable to DoS only

Figure 6-4: Genetic algorithm with a population size of 20 and a generation size of 40. The algorithm searches through HTTP Request smuggling and DoS vulnerabilities in a given application server endpoint. The search space has a size of $n = 1.6 \times 10^{21}$

We do not observe any meaningul differences between both strategies with this grammar. By searching for missions instead of requests and by refining how we formualate requests around the mission, we are shifting away from an unrestricted search in the space of API calls to an unrestricted search in much narrower space of mission definitions. To understand this better, take the difference between an HTTP Request Summing attack and a Denial of Service attack. Denial of Service exploits a functions arguments, while HTTP Request Smuggling sneaks a function as it is defined. In previous experiment, the syntax for both attacks were the same. The GE generated a request with arguments and assigned missions to that request. Even though the algorithm did not need to explore arguments for an HTTP Smuggling Attack, it did so anyways because the search was unrestricted. In this experiment, actions for both HTTP Request Smuggling and Denial Of Service are well defined. The GE will not attempt to generate requests with arguments when it doesn't need to. This results in a better outcome for the stealthy strategy since much fewer requests are penalized overall. Table 6.9 shows the mission results for the campaign against an an engagement environment vulnerable to all attacks. Table 6.10 shows the results for a similar campaign but with a persistant strategy. Table 6.11 shows the mission results for the campaign against an an engagement environment vulnerable to DoS attacks only. Table 6.12 shows the results for a similar campaign but with a persistant strategy. As we can see, both strategies have similar outcomes.

Table 6.9: Mission results for a stealthy strategy. The application is vulnerable to all attacks - Best Solution (Figure 6-4a)

Best Response			
Number of missions	1	**Fitness**	6113.06
Mission List			
id	**Method**	**Action**	**Method Option**
disrupt	denial_of_service	ExpensiveOperation(”@]]”)	Transform:$arg = 2^4 arg$
exfiltrate	http_smuggling	“ForbiddenOperation”	Decoy:”ExpectedOperation”
infiltrate	http_smuggling	“UnknownOperation”	Decoy:”ExpensiveOperation”

Table 6.10: Mission results for a persistant strategy. The application is vulnerable to all attacks - Best Solution (Figure 6-4c)

Best Response			
Number of missions	3	**Fitness**	6180.86
Mission List			
id	**Method**	**Action**	**Method Option**
disrupt	denial_of_service	ExpensiveOperation(”%@5]”)	Transform:$arg = 2^9 arg$
exfiltrate	http_smuggling	“ForbiddenOperation”	Decoy:”ExpectedOperation”
infiltrate	http_smuggling	“UnknownOperation”	Decoy:”ExpensiveOperation”

Table 6.11: Mission results for a stealthy strategy. The application is vulnerable to DoS attacks only - Best Solution (Figure 6-4b)

Best Response			
Number of missions	2	**Fitness**	4144.90
Mission List			
id	**Method**	**Action**	**Method Option**
disrupt	denial_of_service	ExpensiveOperation(”-.]”)	Transform:$arg = 2^7 arg$

Table 6.12: Mission results for a persistant strategy. The application is vulnerable to DoS attacks only - Best Solution (Figure 6-4d)

Best Response			
Number of missions	1	**Fitness**	4423.34
Mission List			
id	**Method**	**Action**	**Method Option**
disrupt	denial_of_service	ExpensiveOperation(”@]]”)	Transform:$arg = 2^4 arg$

6.5 Summary of results

Section 6.1 establishes that a GE can explore HTTP requests and converge to a request with desired response. Section 6.2 demonstrates that GE performs significantly better than random search on finding vulnerabilities in a system. Section 6.3 shows that the GE scales as we expand the breadth of possible attacks and that applying a strategy to the seeded attack improves how quickly the GE can generate an optimal attack. The type of strategy will have different outcomes depending on the defense strategy. Section 6.4 establishes a mission definition and restricts the requests that are generated to fit the mission. The fitness of a mission-based attack is significantly

higher than of those without a defined mission.

Chapter 7

Conclusion and future work

Our results demonstrate that searching through an attack space using simple grammatical evolution algorithm can beat a simple static web application firewall defense. Grammatical evolution also scales as we increase the breadth of attack and improve our grammar to be more expressive towards mission campaigns and strategies. The application of the heuristics developed in this research to create a dynamic defense strategy presents its own set of challenges. For one, a defender must ensure it does not react to legitimate traffic, and so must maintain the false positive rate low[43]. In addition, the cost of failing for the defender is typically higher than the cost of failing for the attacker. Web applications are also typically sensitive to performance so the defender must strike a balance between the defense reaction time and the application runtime.

7.1 Limitations

This research was performed at very small scale. The application and methods developed merely emulate and model real requests and strategies for adverserial engagements in cybersecurity. Another limitiation lies in the runtime of each experiment. Each expirement takes about an hour to run, making repeatablity scale computationally intractable. This could be improved by parallelizing all requests from the same generation.

7.2 Future Work

This experiment serves to motivate the study of dynamic engement between an attack and the cloud WAF. The below items serve to inform areas of improvements in the framezwork and methodologies developed.

1. The attack setup was limited to 3 types of attacks. Future experiments should explore the effect of increasing the breadth of attacks and strategies on algorithmic scalability.

2. The engagement environment in this study remained static and constant. Future research should study the use a dynamic, GE based defense, as well as the effects of coevolution in the attacker-defender dynamic part of this study.

3. All 3 types of attacks were weighted equally, future work can explore the effects of favoring an attack over another.

4. The vulnerabilties in engagement environment favored a bold attack strategy since the WAF did not punish attackers. Future experiments can explore the dynamics of attack detection on the WAF.

5. All experiments were run on a single application server. Future experiements could explore attacks on multiple endpoints and expoloit the weakest one.

Appendix A

Threat Model

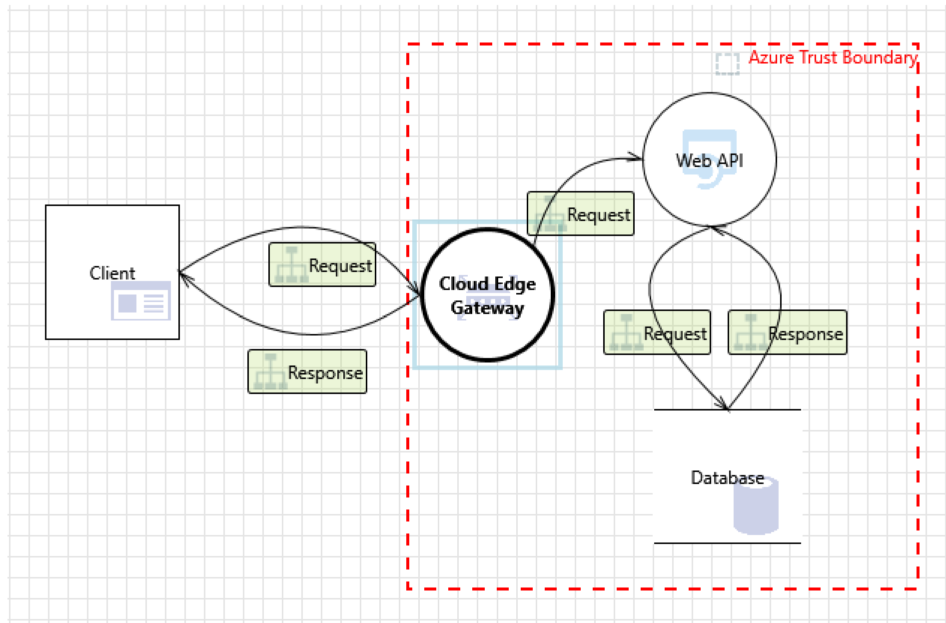


Figure A-1: Threat Model Diagram

A.1 Client to Edge Request

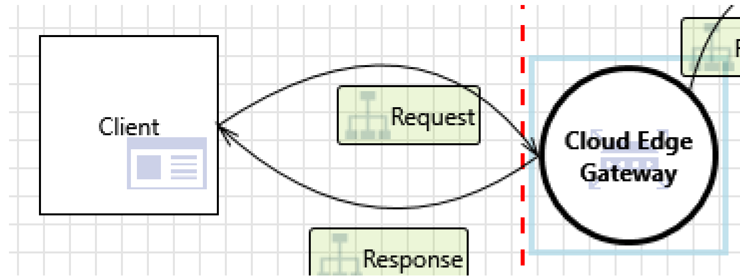


Figure A-2: Theat Model Diagram for client to edge request interaction

Table A.1: An adversary can gain unauthorized access to configure resources.

Category:	Elevation of Privileges
Description:	An adversary can gain unauthorized access to configure resources. The adversary can be either a disgruntled internal user, or someone who has stolen the credentials of a resource manager.
Possible Mitigation(s):	Enable fine-grained access management to Azure Subscription using RBAC.
SDL Phase:	Design

Table A.2: An adversary can deny actions on Cloud Gateway due to lack of auditing.

Category:	Repudiation
Description:	An adversary may perform actions such as spoofing attempts, unauthorized access etc. on Cloud gateway. It is important to monitor these attempts so that adversary cannot deny these actions.
Possible Mitigation(s):	Ensure that appropriate auditing and logging is enforced on Cloud Gateway.
SDL Phase:	Design

Table A.3: An adversary may spoof an system administrator and gain access to the system management portal.

Category:	Spoofing
Description:	An adversary may spoof a system administrator and gain access to the system management portal if the administrator's credentials are compromised.
Possible Mitigation(s):	Enable fine-grained access management to the management portal using RBAC. Enable Multi-Factor Authentication for the Administrators.
SDL Phase:	Design

A.2 Web API to Database

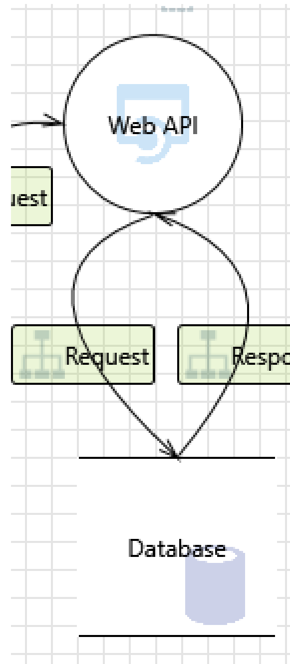


Figure A-3: Theat Model Diagram for Web API to Database request interaction

Table A.4: An adversary can gain unauthorized access to database due to lack of network access protection

Category:	Elevation of Privileges
Description:	If there is no restriction at network or host firewall level, to access the database then anyone can attempt to connect to the database from an unauthorized location
Possible Mitigation(s):	Configure a Windows Firewall for Database Engine Access.
SDL Phase:	Implementation

Table A.5: An adversary can gain unauthorized access to database due to loose authorization rules

Category:	Elevation of Privileges
Description:	Database access should be configured with roles and privilege based on least privilege and need to know principle.
Possible Mitigation(s):	Ensure that least-privileged accounts are used to connect to Database server. Implement Row Level Security RLS to prevent tenants from accessing each others data. Sysadmin role should only have valid necessary users .
SDL Phase:	Implementation

Table A.6: An adversary can gain access to sensitive PII or HBI data in database

Category:	Information Disclosure
Description:	Additional controls like Transparent Data Encryption, Column Level Encryption, EKM etc. provide additional protection mechanism to high value PII or HBI data.
Possible Mitigation(s):	Use strong encryption algorithms to encrypt data in the database. Ensure that sensitive data in database columns is encrypted. Ensure that database-level encryption (TDE) is enabled. Ensure that database backups are encrypted. Use SQL server EKM to protect encryption keys. Use AlwaysEncrypted feature if encryption keys should not be revealed to Database engine.
SDL Phase:	Implementation

Table A.7: An adversary can gain access to sensitive data by performing SQL injection

Category:	Information Disclosure
Description:	SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.
Possible Mitigation(s):	Ensure that login auditing is enabled on SQL Server. Enable Threat detection on Azure SQL database. Do not use dynamic queries in stored procedures.
SDL Phase:	Implementation

Table A.8: An adversary can deny actions on database due to lack of auditing

Category:	Repudiation
Description:	Proper logging of all security events and user actions builds traceability in a system and denies any possible repudiation issues. In the absence of proper auditing and logging controls, it would become impossible to implement any accountability in a system.
Possible Mitigation(s):	Ensure that login auditing is enabled on SQL Server.
SDL Phase:	Implementation

Table A.9: An adversary can tamper critical database securables and deny the action

Category:	Tampering
Description:	An adversary can tamper critical database securables and deny the action
Possible Mitigation(s):	Add digital signature to critical database securables.
SDL Phase:	Design

Table A.10: An adversary may leverage the lack of monitoring systems and trigger anomalous traffic to database

Category:	Tampering
Description:	An adversary may leverage the lack of intrusion detection and prevention of anomalous database activities and trigger anomalous traffic to database
Possible Mitigation(s):	Enable Threat detection on SQL database.
SDL Phase:	Design

A.3 Edge to Web API Request

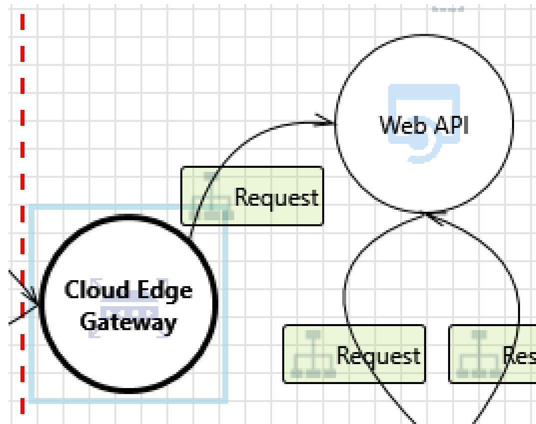


Figure A-4: Theat Model Diagram for Edge to Web API request interaction

Table A.11: An adversary may gain unauthorized access to Web API due to poor access control checks

Category:	Elevation of Privileges
Description:	An adversary may gain unauthorized access to Web API due to poor access control checks
Possible Mitigation(s):	Implement proper authorization mechanism in the Web API Implementation.
SDL Phase:	Implementation

Table A.12: An adversary can gain access to sensitive information from an API through error messages

Category:	Information Disclosure
Description:	An adversary can gain access to sensitive data such as the following, through verbose error messages - Server names - Connection strings - Usernames - Passwords - SQL procedures - Details of dynamic SQL failures - Stack trace and lines of code - Variables stored in memory - Drive and folder locations - Application install points - Host configuration settings - Other internal application details
Possible Mitigation(s):	Ensure that proper exception handling is done in the Web API Implementation.
SDL Phase:	Implementation

Table A.13: An adversary can gain access to sensitive data by sniffing traffic to Web API

Category:	Information Disclosure
Description:	An adversary can gain access to sensitive data by sniffing traffic to Web API
Possible Mitigation(s):	Force all traffic to Web APIs over HTTPS connection.
SDL Phase:	Implementation

Table A.14: An adversary can gain access to sensitive data stored in Web API's config files

Category:	Information Disclosure
Description:	An adversary can gain access to the config files. and if sensitive data is stored in it, it would be compromised.
Possible Mitigation(s):	Encrypt sections of Web API's configuration files that contain sensitive data.
SDL Phase:	Implementation

Table A.15: Attacker can deny a malicious act on an API leading to repudiation issues

Category:	Repudiation
Description:	Attacker can deny a malicious act on an API leading to repudiation issues
Possible Mitigation(s):	Ensure that auditing and logging is enforced on Web API.
SDL Phase:	Design

Table A.16: An adversary may spoof Cloud Edge Gateway and gain access to Web API

Category:	Spoofing
Description:	If proper authentication is not in place, an adversary can spoof a source process or external entity and gain unauthorized access to the Web Application
Possible Mitigation(s):	Ensure that standard authentication techniques are used to secure Web APIs.
SDL Phase:	Design

Table A.17: An adversary may inject malicious inputs into an API and affect downstream processes

Category:	Tampering
Description:	An adversary may inject malicious inputs into an API and affect downstream processes
Possible Mitigation(s):	Ensure that model validation is done on Web API methods. Implement input validation on all string type parameters accepted by Web API methods.
SDL Phase:	Design

Table A.18: An adversary can gain access to sensitive data by performing SQL injection through Web API

Category:	Tampering
Description:	SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.
Possible Mitigation(s):	Ensure that type-safe parameters are used in Web API for data access.
SDL Phase:	Implementation

A.4 Web API to Database Response

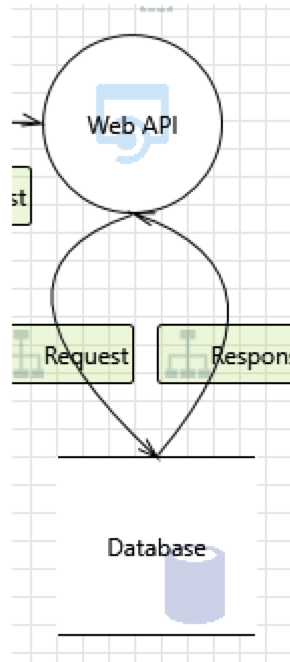


Figure A-5: Theat Model Diagram for Web API to Database response interaction

Table A.19: An adversary may gain unauthorized access to Web API due to poor access control checks

Category:	Elevation of Privileges
Description:	An adversary may gain unauthorized access to Web API due to poor access control checks
Possible Mitigation(s):	Implement proper authorization mechanism in the Web API Implementation.
SDL Phase:	Implementation

Table A.20: An adversary can gain access to sensitive information from an API through error messages

Category:	Information Disclosure
Description:	An adversary can gain access to sensitive data such as the following, through verbose error messages - Server names - Connection strings - Usernames - Passwords - SQL procedures - Details of dynamic SQL failures - Stack trace and lines of code - Variables stored in memory - Drive and folder locations - Application install points - Host configuration settings - Other internal application details
Possible Mitigation(s):	Ensure that proper exception handling is done in the Web API Implementation.
SDL Phase:	Implementation

Table A.21: An adversary can gain access to sensitive data by sniffing traffic to Web API

Category:	Information Disclosure
Description:	An adversary can gain access to sensitive data by sniffing traffic to Web API
Possible Mitigation(s):	Force all traffic to Web APIs over HTTPS connection.
SDL Phase:	Implementation

Table A.22: An adversary can gain access to sensitive data stored in Web API's config files

Category:	Information Disclosure
Description:	An adversary can gain access to the config files. and if sensitive data is stored in it, it would be compromised.
Possible Mitigation(s):	Encrypt sections of Web API's configuration files that contain sensitive data.
SDL Phase:	Implementation

Table A.23: Attacker can deny a malicious act on an API leading to repudiation issues

Category:	Repudiation
Description:	Attacker can deny a malicious act on an API leading to repudiation issues
Possible Mitigation(s):	Ensure that auditing and logging is enforced on Web API.
SDL Phase:	Design

Table A.24: An adversary may spoof Database and gain access to Web API

Category:	Spoofing
Description:	If proper authentication is not in place, an adversary can spoof a source process or external entity and gain unauthorized access to the Web Application
Possible Mitigation(s):	Ensure that standard authentication techniques are used to secure Web APIs.
SDL Phase:	Design

Table A.25: An adversary may inject malicious inputs into an API and affect downstream processes

Category:	Tampering
Description:	An adversary may inject malicious inputs into an API and affect downstream processes
Possible Mitigation(s):	Ensure that model validation is done on Web API methods. Implement input validation on all string type parameters accepted by Web API methods.
SDL Phase:	Design

Table A.26: An adversary can gain access to sensitive data by performing SQL injection through Web API

Category:	Tampering
Description:	SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.
Possible Mitigation(s):	Ensure that type-safe parameters are used in Web API for data access.
SDL Phase:	Implementation

A.5 Client to Edge Response

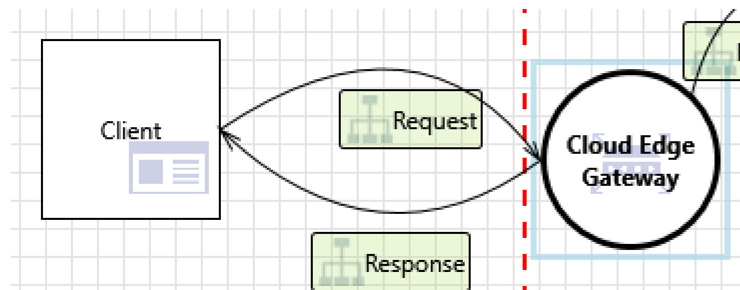


Figure A-6: Theat Model Diagram for Client to Edge response interaction

Table A.27: An adversary may spoof an system administrator and gain access to the system management portal.

Category:	Spoofing
Description:	An adversary may spoof a system administrator and gain access to the system management portal if the administrator's credentials are compromised.
Possible Mitigation(s):	Enable fine-grained access management to the management portal using RBAC. Enable Multi-Factor Authentication for the Administrators.
SDL Phase:	Design

Table A.28: An adversary can gain unauthorized access to configure resources.

Category:	Elevation of Privileges
Description:	An adversary can gain unauthorized access to configure resources. The adversary can be either a disgruntled internal user, or someone who has stolen the credentials of a resource manager.
Possible Mitigation(s):	Enable fine-grained access management to Azure Subscription using RBAC.
SDL Phase:	Design

Appendix B

Listings

Listing B.1: Definition of an HTTP request smuggling OWASP Core Rule Set Rule

```
#
# -=[ HTTP Request Smuggling ]=-
#
# [ Rule Logic ]
# This rule looks for a CR/LF character in combination with a HTTP / WEBDAV method name.
# This would point to an attempt to inject a 2nd request into the request, thus bypassing
# tests carried out on the primary request.
#
# [ References ]
# http://projects.webappsec.org/HTTP-Request-Smuggling
#
SecRule ARGS_NAMES|ARGS|XML:/* "@rx [\n\r]+(?:get|post|head|options|connect|put|delete|trace| \
                                        track|patch|propfind|propatch|mkcol|copy| \
                                        move|lock|unlock)\s+[^\s]+(?:\s+http|[\r\n])" \
    "id:921110,\
    phase:2,\
    block,\
    capture,\
    t:none,t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,\
    msg:'HTTP Request Smuggling Attack',\
    logdata:'Matched Data: %{TX.0} found within %{MATCHED_VAR_NAME}: %{MATCHED_VAR}',\
    tag:'application-multi',\
    tag:'language-multi',\
    tag:'platform-multi',\
    tag:'attack-protocol',\
    tag:'paranoia-level/1',\
    tag:'OWASP_CRS',\
    tag:'OWASP_CRS/WEB_ATTACK/REQUEST_SMUGGLING',\
```

```
ctl:auditLogParts=+E,\
ver:'OWASP_CRS/3.2.0',\
severity:'CRITICAL',\
setvar:'tx.http_violation_score=+%{tx.critical_anomaly_score}',\
setvar:'tx.anomaly_score_pl1=+%{tx.critical_anomaly_score}'"
```

Listing B.2: Grammar definition of a request query expressed in Backus–Naur form (BNF)

```
<request> ::= {"mission" : <mission>, "actions": <actions>}
<mission> ::= ["disrupt"] | ["exfiltrate"] | ["infiltrate"] |
              ["disrupt", "exfiltrate"] | ["disrupt", "infiltrate"] |
              ["exfiltrate", "infiltrate"] |
              ["disrupt", "exfiltrate", "infiltrate"]

<actions>        ::= <allowed_action> | <illegal_action>
<allowed_action> ::= {"Type" : <type>, "Name" : <api_name>, "Args" : <args> }
<illegal_action> ::=  { "Name" : <illegal_name>, "Args" : <args> }
<type>           ::= "post" | "get"

<any_name>     ::= <api_name> | <illegal_name>
<api_name>     ::= "ExpectedOperation" | "ExpensiveOperation" | "ForbiddenOperation"
<illegal_name> ::= "UnknownOperation"

<args>     ::= [] | [<arg_list>]
<arg_list> ::= <arg>
<arg>      ::= "<string>" | <number>

<string>    ::= <alpha_num> | <special> | <alpha_num><special> |
                <alpha_num><string> | <special><string>
<alpha_num> ::= <alpha> | <digit> | <alpha><digit> | <alpha><alpha_num> |
                <digit><alpha_num>
<alpha>     ::= A | B | C | ... | X | Y | Z | a | b | c | ... | x | y | z
<number>    ::= <digit> | <digit><number>
<digit>     ::= 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<special>   ::= ! | @ | # | $ | % | ^ | & | * | ( | ) | _ | - | = |
                + | . | , | { | } | \\ | / [ | ]
```

Listing B.3: Grammar definition of a mission expressed in Backus–Naur form (BNF)

```
<missions> ::= [<disrupt>] | [<exfiltrate>] | [<infiltrate>] | [<disrupt>, <exfiltrate>] |
               [<disrupt>, <infiltrate>] | [<exfiltrate>, <infiltrate>] |
               [<disrupt>, <exfiltrate>, <infiltrate>]

<disrupt>    ::= {"id" : "disrupt", "method" : <denial_of_service>, "action" : <api_action>}
<exfiltrate> ::= {"id" : "exfiltrate", "method": <http_smuggling>, "action" : <api_action>,
                  "expected_payload" : "SECRET_PAYLOAD"}
<infiltrate> ::= {"id" : "infiltrate", "method": <http_smuggling>, "action" : <xscript_action>,
                  "expected_payload" : "HIJACKED_PAYLOAD"}

<denial_of_service> ::= {"id": "denial_of_service",
                         "arg_transform": "lambda x : x*(int)(math.pow(2,<digit>))"}
<http_smuggling>    ::= {"id": "http_smuggling", "decoy_action" : <api_action>}

<api_action>     ::= {"Type" : "get", "name": "ExpectedOperation", "Args" : []} |
                     {"Type" : "post", "name": "ExpensiveOperation", "Args" : [<arg>]} |
                     {"Type" : "post", "name": "ForbiddenOperation", "Args" : []}
<xscript_action> ::= {"Type" : "post", "name": "UnknownOperation", "Args" : []}

<args>     ::= [] | [<arg_list>]
<arg_list> ::= <arg>
<arg>      ::= "<string>" | <number>

<string>    ::= <alpha_num> | <special> | <alpha_num><special> |
                <alpha_num><string> | <special><string>
<alpha_num> ::= <alpha> | <digit> | <alpha><digit> | <alpha><alpha_num> |
                <digit><alpha_num>
<alpha>     ::= A | B | C | ... | X | Y | Z | a | b | c | ... | x | y | z
<number>    ::= <digit> | <digit><number>
<digit>     ::= 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<special>   ::= ! | @ | # | $ | % | ^ | & | * | ( | ) | _ | - | = |
                + | . | , | { | } | \\ | / [ | ]
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Stephen Watts and Muhammad Raza. SaaS vs PaaS vs IaaS: What's The Difference and How To Choose. June 2019.

[2] CloudFlare. What is BGP? BGP routing explained. November 2018.

[3] Cybersecurity Insiders. Cloud security report. March 2019. `https://www.isc2.org/-/media/ISC2/Landing-Pages/2019-Cloud-Security-Report-ISC2.ashx`.

[4] Jennifer Adams and Andras Cser. Forrester analytics: Cloud security solutions forecast, 2018 to 2023. April 2019.

[5] Jay Heiser. Clouds are secure: Are you using them securely? September 2015. Gartner Report.

[6] John Maddison. The hidden challenge of the cloud security skills gap. July 2019.

[7] Josh Fruhlinger. What is a cyber attack? February 2020.

[8] Kevin Rock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving Censorship Evasion Strategies. November 2019.

[9] Victor R. Kebande and Hein. S. Venter. A cognitive approach for botnet detection using Artificial Immune System in the cloud. October 2014.

[10] U. Aickelin, D. Dasgupta, and F. Gu. Artificial Immune Systems. 2014.

[11] National Institute of Standars and Technology. *NIST Cloud Computing Program*. U.S. Department of Commerce, November 2010.

[12] Frank D. Rosa, Mary J. Turner, Deepak Mohan, Larry Carvalho, David Tapper, William Lee, Adelaide O'Brien, and Richard L. Villars. IDC Futurescape: Worldwide Cloud 2020 Predictions. October 2019.

[13] Floyd Piedad and Michael Hawkins. *High availability design, techniques and processes*. Prentice Hall, 2001.

[14] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. *Proceedings of the 17th international conference on World Wide Web. ACM*, 2008.

[15] Biharck Muniz Araujo. *Hands-On RESTful Web Services with TypeScript 3*, section 3, pages 60–85. RESTful Web Services. Packt Publishing, March 2019.

[16] Rapid API. Getting started with rapid API.

[17] Cisco. BGP Best Path Selection Algorithm. September 2016.

[18] Li Long, Ma XiaoZhen, and Huang Yulan. CDN Cloud: A novel scheme for combining CDN and Cloud Computing. August 2013.

[19] Trustwave Global. 2020 trustwave global security report. April 2020.

[20] Christian Folini, Walter Hop, and Chaim Sanders. Owasp modsecurity core rule set v3.2. `https://github.com/SpiderLabs/owasp-modsecurity-crs/tree/v3.3/dev/rules`, September 2019.

[21] Christian Folini and Ivan Ristic. *ModSecurity Handbook*. Feisty Duck, second edition, July 2017.

[22] Amazon AWS. Managed rules for aws web application firewall. `https://aws.amazon.com/marketplace/solutions/security/waf-managed-rules`, September 2015.

[23] Microsoft Azure. What is azure web application firewall? `https://docs.microsoft.com/en-us/azure/web-application-firewall/overview`, August 2019.

[24] Google Cloud. Understanding google cloud armor's new waf capabilities. `https://cloud.google.com/blog/products/identity-security/understanding-google-cloud-armors-new-waf-capabilities`, November 2019.

[25] David A. Wheeler. *Secure Programming*, chapter 2.4. 2015.

[26] J.W Backus, F.L. Bauer, J.Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, and J.H. Wegstein. *Revised report on the algorithmic language ALGOL 60*. Communications of the ACM, January 1963.

[27] Michael O'Neill and Conor Ryan. *Grammatical evolution*, volume 5 of *IEEE Transactions on Evolutionary Computation*, pages 349–358. IEEE, August 2001.

[28] Una-May O'Reilley, Jamal Toutouh, Marcos Pertierra, Daniel Prado Sanchez, Dennis Garcia, Anthony Erb Lugo, Jonathan Kelly, and Erik Hemberg. *Adversarial Genetic Programming for Cyber Security: A Rising Application Domain Where GP Matters*. Genetic Programming and Evolvable Machines. Springer, 2020.

[29] Micrsoft. Microsoft threat modeling tool. `https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool`, February 2017.

[30] Li Jiang, , Hao Chen, and Fei Deng. A security evaluation method based on STRIDE model for web service. May 2010.

[31] Marget Rouse. Advanced persistent threat (APT). https://searchsecurity.techtarget.com/definition/advanced-persistent-threat-APT, November 2010.

[32] Liviu Arsene. The Anatomy of Advanced Persistent Threats. https://www.darkreading.com/partner-perspectives/bitdefender/the-anatomy-of-advanced-persistent-threats/a/d-id/1319525, March 2015.

[33] U.S. Senate-Committee on Commerce Science and Transportation. A "kill chain" analysis of the 2013 target data breach. March 2014.

[34] James Forshaw. *Attacking Network Protocols*, chapter 9. No Startch Press, August 2017.

[35] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol. December 2011.

[36] Robert Auger. Http request smuggling. December 2009.

[37] Equifax. Equifax Releases Details on Cybersecurity Incident. https://www.equifaxsecurity2017.com/2017/09/15/equifax-releases-details-cybersecurity-incident-announces-personnel-changes/, 2017.

[38] MITRE. *CVE-2017-5638 Detail*. U.S. Department of Commerce, March 2017.

[39] Mathew J. Schwartz. Equifax's Data Breach Costs Hit $1.4 Billion. May 2019.

[40] Erik Hemberg. Donkey GE. https://github.com/flexgp/donkey_ge, 2018.

[41] Eric Knorr. What serverless computing really means. July 2016.

[42] Robert Axelrod and Rumen Iliev. Timing of cyber conflict. January 2014.

[43] Ericka Chickowski. Every Hour SOCs Run, 15 Minutes Are Wasted on False Positives. 2019.