Status Report
Cara Corporation Simple Point of Presence Web Server
August 17, 2015

This document briefly summarizes the key pieces of information for iSEC Partners' assessment of Cara Corporation's Simple Point of Presence Web Server. The status report discusses the following items:

1 Overview

My task was to design a webserver that served the output of an IoT device onto the network; bearing in mind the security implications of that task.

This project required me to don two hats. One, the developer; and two, the penetration tester. I used Python to build a webserver that responded properly to GET requests as per RFC 2616 ,[1] and I then built a web client atop it using Flask. Lastly, I took the liberty to explore some possible vulnerabilites in the Flask framework itself.

[1] https://www.ietf.org/rfc/rfc2616.txt

2 Task Status

Project Logistics

Task	**To Start**	**In Progress**	**Complete**	**Notes**
Create repo for project		●		Codebase prepped - upload pending GitLab access
Develop a web server that implements the HTTP GET method on a user specified port			●	
Have server return proper headers			●	
Document all security related issues - vulns discovered, imagined, remidiated			●	
Have server return a local file			●	
Make sure server is accessible via a modern web browser			●	

3 Schedule

Project Schedule					
Week 1: August 3rd– August 7th					
	Monday	Tuesday	Wednesday	Thursday	Friday
Task	Socket/WSGI server research	Cursory server implementation	Further server implementation / **Touch base with client**	Gone to DEFCON	Gone to DEFCON
Week 2: August 10th– August 14th					
	Monday	Tuesday	Wednesday	Thursday	Friday
Task	Flask / Jinja research	Outline web front-end	Actualize outline using Flask / Jinja	Testing / Bringing features closer to spec	Final testing
Week 2: August 17th– August 21st					
	Monday	Tuesday	Wednesday	Thursday	Friday
Task	**Status Meeting** / Ship code				

4 Key Contacts List

4.1 Internal and External Teams

The iSEC team has the following primary member:

- Jack Leadford — Application Security Engineer Intern
  Jack.Leadford@nccgroup.trust, (510) 717–9380

The Cara Corporation team has the following primary members:

- Cara Marie — Cara Corporation
  Cara.Marie@nccgroup.trust, (818) 856–8344

5 Security Findings

5.1 Classifications

The following section describes the classes, severities, and exploitation difficulty rating assigned to each identified issue by iSEC.

Vulnerability Classes

Class	Description
Access Controls	Related to authorization of users, and assessment of rights
Auditing and Logging	Related to auditing of actions, or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to mathematical protections for data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to the race conditions, locking, or order of operations

Severity Categories

Severity	Description
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small, or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, of moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation or serious legal implications

Difficulty Levels

Difficulty	Description
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

1. Taking advantage of a large asset

Class: Denial of Service **Severity:** Informational **Difficulty:** Undetermined

Finding ID: iSEC-POD-1

Targets: 404 Page - Served whenever a path/file is requested that isn't explictly routed (often)

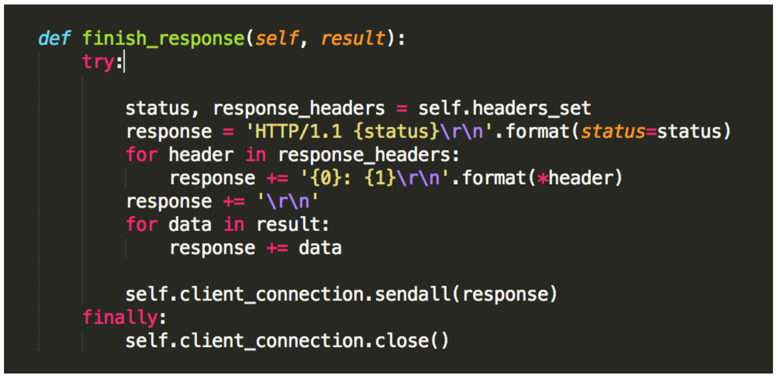
Description: The 404 page contains the single largest asset on the site: a 400KB .gif. Realizing this, I wrote a quick script to request said page a heinous number of times:

```
import urllib2
print 'Please enter in the port that the server is being run on:'
input_port = int(raw_input())
print 'Please enter in the page you would like to hammer: \n e.g. /files'
input_path = raw_input()
num = 0
while True:
num = num +1
urllib2.urlopen("http://localhost:" + str(input_port) + input_path).read()
print "GET Requests: \n", num
```

Interestingly enough, when let run into the 10/20,000 total requests range, the DoS tool would break if I requested the same page in a browser (it was served to the browser with zero issue). It seems like the server grants a higher affinity to a GET from a browser. Thus eliminating a situation where the DoS tool is sending requests to the single thread faster than a browser can get to it; effectively monopolizing the thread and preventing any other requests from going through.

Further testing is needed to establish exactly why this is the case.

Also, the server was completely insusceptible to a Slow Loris type of a attack; wherein many partial HTTP requests are made to a server in an attempt to open as many connections as possible and overflow the server. I suspect a similar causation as the ineffectiveness of the tool above. The simple logic of the server has proven to be a fairly good DoS defense.



```
def finish_response(self, result):
    try:

        status, response_headers = self.headers_set
        response = 'HTTP/1.1 {status}\r\n'.format(status=status)
        for header in response_headers:
            response += '{0}: {1}\r\n'.format(*header)
        response += '\r\n'
        for data in result:
            response += data

        self.client_connection.sendall(response)
    finally:
        self.client_connection.close()
```

Figure 1: Closes connection NO MATTER WHAT - seems fairly effective in practice

My final thought was to try and hook the server's single thread, but as the application is in its nascent stages, I was unable to find a vector for this (besides writing/trying the DoS tool, but that was not fruitful).

Exploit Scenario: A malicious user executes a DoS/DDoS against the system; crashing the server. Unfavorable outcomes are the result of the domino effect of: →DoS attack →reliability compromised →customers upset →increase in support calls / loss of business / reputation takes a hit

Short Term Solution: Use a smaller asset, perhaps a small .jpg/static text, instead of the .gif on the 404 page.

Long Term Solution: Implement some actionable rules in IPtables/Fail2Ban that limit the amount of requests a single host may make to something sane. Also, if the box doesn't need to be routed over the internet, make sure it's not. Rule lists can be provided if need-be.

2. Sandbox Shmandbox

Class: Data validation	**Severity:** Informational	**Difficulty:** Undetermined

Finding ID: iSEC-POD-2

Targets: Parts of a Jinja2 templates where logic is run based on user input/interaction

Description:

Jinja uses Python logic to generate dynamic content. And just like your run of the mill HTML/JS XSS, this has the potential to be exploited as a path to executing arbitrary code.

Pulled straight from the Jinja documentation:

> Jinja1 was running sandbox mode by default. Few applications actually used that feature so it became optional in Jinja2. For more details about the sandboxed execution see SandboxedEnvironment.

And now, Flask's mention of Jinja's sandbox mode on their homepage:

> Sandboxed execution mode. Every aspect of the template execution is monitored and explicitly whitelisted or blacklisted, whatever is preferred.

This ambiguity/incongruence is concerning because it's presented as something that points developers towards not needing to perform their due dilligence. And this is especially concerning with Jinja, as most of the tools for checking if an attribute/function/method is safe are contained within it. Breaking out of a sandbox is now moot, because there isn't one.

You might say that this type of attack would most certainly be thwarted by input validation and, while Flask invokes Jinja's autoescape extension by default, its standard ruleset is flimsy at best. Plus, their language in terms of that is also very affirmative in terms of your safety position if you have it enabled. I'm sure this has many a developer has established a dangerous reliancy on the tool and made unbased assumptions about the security of their code without having had tested it. So in that sense it's concerning in the same sense as the lack of sandboxing: inherently worrisome, sure, but more because of how it's presented to developers in a way that encourges brushing it off.

It seems then that we can form a hypothesis that many Flask/Jinja2 deployments A) aren't sandboxed, B) have flimsy escaping of HTML characters at best, and C) execute some sort of Python based around/on user input. Moving forward, attempting to perform the equivelent of a blind SQL-injection on the template logic seems to be an interesting path.

Exploitation of this kind would require knowledge of the Flask/Jinja source that likely goes beyond what can be inferred, but nevertheless, something to consider if you were deploying Flask/Jinja out onto the open internet. Recent research shows that this scenario isn't as farfetched as one would think. [2]

Exploit Scenario: A malicious user uses his knowledge of how a system's templating logic works, gained through documentation/trial and error, and uses it to craft a payload that results in arbitrary code being evaluated by the non-sandbox python code within the template. This could easily result in an RCE.

Short Term Solution: Do a general audit of how you're handling user input, especially when it comes to content that operates hand-in-hand with template logic. Implement an HTML attribute WHITELIST - only allow what /you/ need - much more effacious than trying to block every attribute a hacker could exploit, as they'll write payloads faster than you can block them. Is there anywhere where you blindly mark user input as safe? If so, why?

[2] blog.portswigger.net/2015/08/server-side-template-injection.html

Long Term Solution: Make sure you're treating a sandbox like a sandbox (even if it's not). That means being aware of where the Python code is being run (system permissions, users/groups, catching compiler exceptions) and limited the access it has to resources shared with other parts of the system. Also, you could look into deploying the applicaiton in a Docker container; effectively sandboxing it yourself.