**Invoking Remotely Stored Fileless Malware through the Usage of Malicious Macros on a Microsoft Word Document**

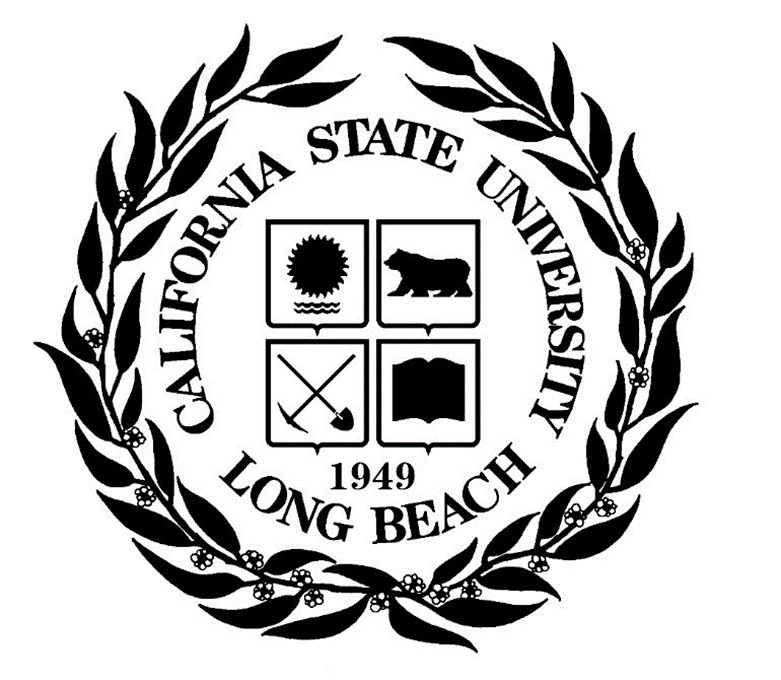
Group 7 - CECS 378 - Fall 2022

Members:

Sean Collins, Sara Hamidi, Aaron Garcia, Matthew Miller, Garrett Towner, Chandler Sidars, Frederick Peralta

Professor:

Louis Uuh



California State University, Long Beach

College of Engineering

11/17/22

**Introduction**

First, I want to talk about how our malware works. Then, I am going to explain why it’s fileless. Finally, I am going to discuss ways that it can be improved.

Our encryption code is located on our website. On the website, there is a button to download a .exe which contains a word document with macros and two .bat files. The .bat files place the word document with the macros in a specific location in the user’s file system. We put the word document in a trusted folder to prevent the antivirus from erasing the word document with the macros in it. The macros in the document open powershell and paste the encryption code from the website in powershell. The user’s computer gets encrypted. The keys are created and sent to our email for decryption purposes. A pop up shows up asking the user to email us if they want their computer decrypted. Finally, we used our decryption code and the emailed key and IV to decrypt the user’s files. Currently, our virus is programmed to only encrypt and decrypt files that are located in the user’s folder.

This project is an example of fileless malware because the functional part of the ransomware is not located on their harddrive. Because fileless malware lives off the land, it uses the computer’s existing temporary memory and runs via an existing program on their computer i.e. Powershell. Additionally, the ransomware is never put on their computer as a .txt or .cpp. There is no runnable code left on their machine except for the initialization programs (the macros).

Currently, if a user runs into our website and clicks the download button, they would need to run the .exe as administrator, opt to keep the file after a warning pop up, and verify a second time to let it make changes to the harddrive. To improve this, we would use a script injection or a dll reflection to make a trojan that acts more like a trustworthy file and needs less user interaction to work. Additionally, deleting the .bat files and the .docm would be another good improvement to avoid leaving evidence on their computer. The reason being, that it would give them insight on how the attack is being done and prevent us from doing it again. Another improvement would be to turn our encryption algorithm into a command prompt script or .exe to take in keyboard arguments instead of being hard coded every time. Finally, automating our email system to decrypt a victim’s files after making their payment would make the decryption process easier and faster.

**Summary of Contributions**

1. Sean Collins
   1. Created a rough class diagram, gathered initial resources, documented step by step tasks for group members.
   2. Coded prototypes for word document macro and website html. Found delivery site for .docm
   3. Programmed encryption and decryption method, WinForms popup, Powershell wrapper, file traversal algorithm, and download exe.
2. Sara Hamidi
   1. Figured out how to email the key and IV to ourselves using Amazon SES
   2. Worked with Sean on creating the initial schedule for the project. Researched useful links and websites that people could use to start their parts. Divided up the parts and came up with rough deadlines for each part.
   3. Wrote the outline/summary of the project in the write up.
3. Aaron Garcia
   1. Took the starting knowledge of Chandler's trial and error in creating the AWS EC2 instance and S3 bucket, and created new AWS resources from scratch to finish Lab 2. The byproduct of that was the completed website, which hosted and displayed the HTML files through a domain name.
   2. Repurposed that same domain and website into the host location of the text file that contained the scripts needed to execute the malicious encryption algorithm.
   3. Modified the macros of the word document to call the encryption script from the text file over the internet, rather than from a file stored locally. This in large effect completed the “fileless” step of the implementation of the “fileless malware.”
4. Matthew Miller
   1. Took the role of designing the website which included the download for the malicious document as well as how the html/css/javascript interacts to download and run the scripts in powershell.
   2. Researched the optimal format for which the malware should be transported to an unsuspecting victim (fake word doc). As well as testing the scripts and determining which datapath the ransomware should follow for the attack.
   3. Worked with Fred to make the PowePoint presentation and assign group members roles and times for the presentation.
5. Garrett Towner
   1. Took the role of designing the Windows pop-up which notifies the user of the encryption of their files and demands a ransom for their decryption. After trying and failing with C#, ended up using powershell commands to accomplish it.
   2. Attempted to make the message font size large by using ASCII representations of the letters, but could not fix the kerning in the Windows pop-up to make it readable.
   3. Collaborated with Sean on the decryption algorithm. Had to make modifications to the encryption program to save the file extension of the encrypted files so that when they are decrypted they are given the proper file extension.
6. Chandler Sidars
   1. Spearheaded development of the Amazon Web Services EC2 instance, establishing the initial configuration which would be used as the foundation for the Apache web server.
   2. Configured numerous aspects of the AWS EC2 instance, such as access control, user privileges, and the network firewall.
   3. Collaborated with Aaron to work through numerous roadblocks associated with the overall role of the EC2 instance within the project’s scope.
7. Frederick Peralta
   1. Tested the malicious script on a virtual machine, traversing the file system for irregularities.
   2. Figured out a solution to extract an identification number from virus code, which needed to be excluded from encryption. This would be utilized by Sean for decryption.
   3. Collaborated with Matt to assemble a PowerPoint presentation.

**Individual Contributions**

**Sean Collins**

The first step of the process was meeting with the professor to get a concrete idea of the functions involved in fileless malware. We began toying with the idea of base 64 command line scripts and bytecode injections using hex. I figured I’d start simple by trying to get ransomware to run as a powershell script. One of the first resources I found used an exploit (really more of a feature) in Internet Explorer to run Visual Basic code (Studebaker, 2017). Sara and I created a work schedule and broke the project into modules with the objective of launching Powershell malware with Visual Basic. While some of the group went to work on a website, I started coding the encryption program. It needed several parts: a file traversal method, an encryption method, an automated email, a popup, and a wrapper to tie everything together. The wrapper was the first and easiest step, completed with a basic script (Furmanek, 2016). File traversal turned out to be simple to code myself with built in C# libraries. It did need a few iterations since running the program recursively on every file turns out to be very slow. I was satisfied with encrypting only the contents of C:\Users. Obviously, in a real attack we would want to target external drives and cloud storage folders. The framework for external drive encryption was included but not thoroughly tested since Windows VMs tend to have only one drive. Encryption turned out to be a bit finicky, and needed to be put in a try/catch block in case the program tried to encrypt a .dll. The C# AES library provided the AES algorithm, which became more reliable with the inclusion of blocks (Dotnet-Bot, n.d.a). Making a popup was a trivial part of the Powershell script made with WinForms (Hicks, 2014). The email stumped me, see Sara’s portion for more information. Last, I reverse engineered the encryption algorithm to make a decryption algorithm. In its current state the keys need to be hard coded before sending over email.

Around this point in development Matt brought it to my attention that no common browsers support Visual Basic anymore. After a bit of panic, I made the concession to use a macro enabled word document as the delivery method. I’m still not thrilled about it, but it was the fastest to develop for testing purposes. I wrote an only slightly modified version of the original Visual Basic code in the macro that depended on a locally stored powershell script. Eventually, Aaron modified this to use an online script via our web server. After a bit of experimentation, it became clear that restrictions on .docm files prevented them from being a viable standalone tool. Many attempts to send it through email were thwarted by antivirus. It would often run once on a computer and never again. In the eleventh hour, I developed a .exe with IExpress that contained two .bat files and the .docm. One .bat moves the .docm into a trusted folder, the other runs the .docm in its new location. These could have easily been one file but were separated for testing purposes. I also attempted to delete the installed files but never got it working properly.

Due to time constraints, poor group coordination, and last minute surprises we were stuck using .exe, .docm, and .bat files that leave a sizable paper trail. In my opinion, we lost the advantages provided by fileless malware in our delivery method. While our demonstration of Powershell vulnerabilities succeeded, we probably should have investigated other angles of attack. Some potentials like script injection and reflective .dll would make for a better implementation.

**Sara Hamidi**

My first job for this project was to create an outline for the tasks and deadlines for each task. I worked with Sean to do this. We created a draw.io and divided the tasks up into different boxes. We googled videos and searched up articles that our group members could refer to get started. We also did this so that they would have an idea of what we were looking for. We finished the initial model around the second week of school. Throughout the project this model obviously changed to adapt to our project. I believe there is at least one version of it that you can take a look at in our submission.

To resolve the email issue, I tried to use OpenSSL to send the key and IV from our victim’s computer back to ourselves. OpenSSL is pre-installed on most Linux/Unix operating systems but has to be installed on Windows. Because we used Windows’s powershell to run the encryption code, and I didn’t have much knowledge on OpenSSL and how it works we were worried that we weren’t going to be able to send an email using OpenSSL if the victim didn’t already have the program pre-installed on their computer. Regardless, I did install and mess around with OpenSSL before searching for another way to send the keys. I used this command “starttls - openssl s\_client -starttls smtp -connect smtp.gmail.com:587 -crlf” to try and connect to gmail. Then I used the command EHLO to identify myself to the smtp server and then I had to use the AUTH PLAIN command to input my credentials. This is the part where I got really stuck. It took me a while to figure out that it wanted my credentials in this format “\0username\0password” converted into base 64. I proceeded to do that but then I would get two different errors depending on the port I tried to connect to. If I tried to connect to port 465, it would say something along the lines of failed to connect. If I tried to connect to port 587, it would continue to say wrong user credentials. I later found out that the reason why it wouldn’t connect to port 465 is because it is the deprecated standard of smtp. I also later learned that the TLS in starttls means that the data being sent is encrypted and that TLS is the successor of SSL. Because we thought we had to have OpenSSL installed to use it and I was stuck at the step where I had to put in my credentials, I tried to see if I could send the emails directly from powershell. It seemed like I could and it looked a lot simpler than the way I was trying to do it before, but I found out the Send-MailMessage command was deprecated. I read somewhere that the reason why I was having trouble with sending the email using OpenSSL had to do with my email settings. I installed Mozilla Thunderbird to change my email settings. I took away two factor authentication because I thought it was affecting my connection. I also tried to find a way to turn off my settings that block third parties from sending me emails. At this point, I realized that I probably should create a burner email for this project. I got stuck trying to figure out how to not enable receiving emails from third parties. I realized that gmail no longer gives its users this option. I started looking into emails that offer this feature. I also made a yahoo mail account thinking it would allow this, and I had the same problem. I found out that Amazon AWS has a service called Amazon SES (Send Email Service). Without paying any extra fees, I was allowed to send out 200 emails a day and 62,000 emails a month without getting charged. I looked up a youtube video on how to start using it. It was pretty easy and straightforward. I sent a test email through their site and it worked so I implemented the sendMail method in our encryption code and that also worked. To implement it I just had to specify the port, the smtpClient, and basically input the credentials amazon gave me. (I was not responsible for the file ID stuff in that method) Everything was going smoothly regarding the emails until we ran into issues sending the document with the macros on it to one another. One of us accidentally used an old version of our code to send the emails in the process of trying to resolve this issue and my inbox got flooded with emails all at around the same time. In addition to the 200 emails a day quota, amazon has a policy where users are not allowed to send a bunch of emails at the same time because they assume that you are spamming someone. As a result, the encryption algorithm was failing to encrypt because the email service was sending an error that I had reached my daily quota for sending out emails. I was concerned that the quota wouldn’t reset in time or at all because they stated on their website that there isn’t a counter that keeps track of the number of emails you sent in a day that resets at a certain time. I didn’t want to risk the quota not resetting properly, so I decided to create a new burner gmail and aws account to send the emails from. Thankfully, the email mechanism is up and running again.

**Aaron Garcia**

During Lab 2, the class was tasked with creating a webpage and pointing a user-friendly domain name to the IP address our website was hosted, in addition to understanding how ports work and how to set up an encrypted connection. That website also contributed to one of the major components of our semester-long project, which was the implementation of the fileless portion of our project, fileless malware.

Weeks later when it was time to place the script in the remote location and invoke it, I did exactly that- I placed the text file in the EC2 instance and called the script remotely.

Getting to that point, however, was deceptively simple.

Let's start with the part of Lab 2 that contributed the most to the semester project: the AWS instance. Right after its creation, I needed to find a way to SSH into it through port 22. PUTTY wasn’t working with either mine or Chandler's AWS instance, but that was when Chandler introduced me to Bitvise. It was free, newer than PUTTY, and much easier to use. It also offered me access to the SSH protocol which allowed me to interact with the server's terminal, and the ability to perform SFTP, which stands for the SSH File Transfer Protocol. These two functionalities were instrumental in allowing me to manipulate and move files.

After Chandler started Lab 2 and provided me with a lot of the tools that made this possible, I decided to create my own AWS instance to understand more of what was going on.

Once the file transfer was possible, I needed a way to start the website and display the HTML files. I then installed and configured apache, which allowed me to access the URL of the webpage, which was required for Lab 2. That was, once I could figure out the permissions of how to allow the files to be transferred from the S3 bucket over to the EC2 instance, which was very painstakingly long. Once all of this was finished, the website worked perfectly, and attaching the domain name was not too much of a challenge either. The result of these steps had the positive side effect of allowing access to the text-based contents of any other files stored on the server, which will be important for me to utilize later on.

At this point, I realized how closely related the requirements for Lab 2 and the creation of the remote host for the malicious script were, so I continued to finish Lab 2, and provide the write-up for its requirements.

Weeks later, I got in contact with Sean who provided me with a PowerShell testing script that simply printed out “Hello World,” the Word Document payload with the malicious macro, resources on how to host the PowerShell script remotely, and some resources that could give me clues on how to call the script. Before I did anything, however, I downloaded a Windows image and created a Virtual Machine to work in, that way I wouldn’t accidentally disturb my actual computer.

I transferred the testing script (script.ps) to the EC2 instance using SFTP via Bitvise. From there, I moved the file into the directory where the webpage was being hosted (/var/www/html) using the EC2 console, just like I did with the HTML file during Lab 2.

From there, my job was to convert the local call into a remote call. I began to edit the Visual Basic code located in the Macros section of the Word Document. Visual Basic can leverage PowerShell to perform the necessary calls to invoke the malicious PowerShell behavior, the question was: how did it need to be written? A tutorial that I was provided with told me to perform a “web-invoke” call to read the plaintext contents of an HTML page, assign the plaintext into a variable, and invoke the variable containing the text as a PowerShell command (Studebaker, 2017.) Since all it was doing was grabbing the plaintext of the specified URI, I instead opted to use a text file rather than the contents of an HTML page. I took the testing script file he provided me with (script.ps) and converted it to a .txt file (script.txt). I took the script the video provided and modified the URI to be one I used during lab and the location of the text file I just modified, and it ran with no issues. A few days later, Sean finished the malware, and I replaced the testing script with the real deal. All that was left was to perform the demonstration, and the project would be complete.

**Matthew Miller**

During the semester project I mainly filled in for what other people did not want to do. The first and main thing that I contributed was the website for which the malware would be hosted. I had never done any html, javascript, or css before this project. Sean helped me out with a basic skeleton of how the website should work and I set up the rest so that the file would be downloaded correctly. The website itself is not very visually appealing since I’m still just starting out, but it does its function well which is what really matters. I had to change some of it around when I figured out that our initial plan for opening powershell would not work. At first we saw some video examples of people using a fake link that would open powershell from the browser (which was our original plan). One of these videos had a command to open powershell by bypassing the is-user check. We then modified this for use in our own website. However, after many hours of trial and error I found out that this was only possible on Internet Explorer, which has now been updated to Edge and naturally made this method impossible.

While looking into how to make the link open powershell, I saw a video of someone using a word doc with macros in it instead. This method still works and is what we used in our final version of the ransomware. Essentially the word doc has a macro inside of it that opens and runs powershell and executes a single command. The command itself tells powershell to read and execute whatever is on an html file. So in our case we have a separate html file that is basically acting as a .txt file which holds our script to encrypt the victim’s files. Once we started testing the malicious word doc, we found that windows defender started to pick up on the document and would not let you download it after a few attempts. We kept changing the encryption slightly and it would work but for consistency we changed the file that the website has into an .exe that would automatically download the word document without having to go through the windows anit-virus. So the final version of the website has a download called coolInstaller.exe that automatically downloads and starts the word document.

**Garrett Towner**

For the semester project I mostly collaborated with Sean on aspects of the malware dealing with the victim’s end. It was surprisingly difficult to get a simple OS pop-up window to work. Initially I tried using C# to do it, but after several hours of even just getting a C# project up and running (it was the first time I ever used C#), it was unclear which library I needed to include to make the kind of pop up window we needed, and it looked like the kind we wanted actually was no longer supported. Then we went searching online for a powershell script to accomplish the same thing. We found one, and I edited it so that it would have no other buttons besides the mandatory close button and that it would display the message threatening the victim and demanding a ransom. The font size was almost comically small, so I tried generating an ASCII art version of the message to make it much bigger so as to grab the victim’s attention, but the pop up window apparently does not have even spacing or applies some annoying kerning to the text it displays which ruins the ASCII message formatting as formed in an even spacing notepad.

I also helped with the decryption algorithm. We took the framework from the encryption algorithm and basically applied it directly with the inverse operation of decrypting. However, we had to change some of the arguments passed to the directory crawling function so that the key would be available and also that the files would be restored to be the proper file name extension. We also had to go back to the encryption algorithm and modify it so that it preserved the file name extension and merely tacked on .enc to the end of the file name, as before that point the file name extension was being overwritten and that information would be difficult to restore any other way than just keeping it the whole time. Also, initially the decryption algorithm was not crawling through the whole directory due to a bug in the code, so we had to troubleshoot through that to find where we introduced the error. Once that was fixed the algorithm was up and running fine.

**Chandler Sidars**

In order for the fileless malware to successfully work, we needed to employ a malicious script that would vector through a web page. The team collectively agreed that the best avenue of approaching this would be through the utilization of an Amazon Web Services EC2 instance to host the web server. I worked closely with Aaron throughout the project’s life cycle in establishing, deploying, and maintaining the EC2 instance to successfully act as the foundation for our team’s project. Initially, I was tasked with getting the EC2 instance configured and running as a prototype to get a rough understanding of how the EC2 instance and the website would be configured.

During the initial setup of the EC2 instance, the majority of my time went to configuring the instance to be accessible by the whole group. As such, my role expanded into utilizing AWS Systems Manager, Identity and Access Management (IAM) roles, users and policies, security groups, and disseminating key pairs (to Aaron before we rotated to Systems Manager. AWS Systems Manager was leveraged to streamline connections to the instance through the use of the session manager. Session manager centralized access control and allowed for team members to connect to the instance either with a browser-based shell or the AWS Command Line Interface (AWS CLI). As a result efficiency was raised significantly as team members didn’t need to use SSH keys and had to wait to have their IP whitelisted. As this was my first experience with AWS, I ran into roadblocks frequently. More often than not, I took advantage of the resources on Youtube and AWS’s extensive documentation on EC2 instances. When I ran into an issue that online resources could not help me due to our project’s use cases, I would collaborate with Aaron and rely on his computer science background to ultimately find a solution that aligned with our project.

**Frederick Peralta**

After Sean completed writing the encryption algorithm and moved his efforts to completing the decryption, he gave me the task of taking that code and testing the code extensively. After grabbing a copy, I booted up a fresh Windows virtual machine and loaded in various programs that the average user would be in their possession. This included various web browsers such as the included Edge browser, as well as Google Chrome and Brave, as a line of communication between victim and attacker is a crucial piece. Opening up an instance of Powershell, I pasted in the code and let it execute. The running time was extensive, but the script executed what it needed to do. After its completion, I poked around various directories and looked at various files to make sure that they were encrypted. All files that were picked up by the encryption had a new file extension appended to them. Unfortunately, the web browsers and any icons present on the taskbar were caught in the encryption algorithm. I notified Sean of this hiccup, and was told that he would take care of it. Unbeknownst to me, whilst also finding a way to exclude the file explorer and web browsers from encryption, he also refactored the code to run significantly faster. After running the new version, it was immediately clear that the taskbar icons and the icons of the Desktop were free from encryption, preserving critical communications.

The malware was designed to encrypt files for financial gain from numerous individuals, so a way to identify and be able to contact them is essential. In the C# code there was a segment that would be assigned a random number used to identify the user’s machine. Another task given to me was to find a way to place that number into a text file, and place that text file on the Desktop. Searching through the documentation of C# by Microsoft, I discovered the two classes that were needed to complete this instruction. The two classes were the *File* and *Environment.* The *File.WriteAllText(<name of file>)* and *Environment.GetFolderPath(<name of directory>)* were the two specific methods that were used from their respective classes. The first was used to write the number to a text file (Dotnet-Bot). The second was used to place the file into the designated directory. I thought it was going to be as simple as writing the text file to the Desktop, but this method is called before the encryption takes place. The text file is run through the algorithm and becomes unreadable. It seemed that there would be no safe folder or directory to place the file to escape the encryption. I decided to look through the C# code, as I remembered that the new code that Sean had provided had prevented the browsers and taskbar. He probably skipped over a folder to keep them from being encrypted. I found that he excluded the hidden *AppData* folder from encryption and had found my safe directory. I created and used the *GetFolderPath* method to place it inside. Every machine has unique user and usernames, so I could not use the absolute path to the directory. The parameter used in this method, *Environment.SpecialFolder.<folder>* would get the folder path regardless of username (Dotnet-Bot). After the algorithm finished encrypting all designated files, it was now safe to move the text file to the Desktop. The user was now in possession of the identification number needed for the attacker to decrypt the files.

**References**

3bdalla, Archemar, & user358915. (1962, December 1). *Send email using openssl*. Unix & Linux Stack Exchange. Retrieved November 17, 2022, from <https://unix.stackexchange.com/questions/236941/send-email-using-openssl>

Ardal, T. (2022, September 20). *How to send emails from c#/.net - the definitive tutorial*. elmah.io Blog - .NET Technical tutorials/guides and new features. Retrieved November 17, 2022, from https://blog.elmah.io/how-to-send-emails-from-csharp-net-the-definitive-tutorial/

Dotnet-Bot. (n.d.). *AES class (system.security.cryptography)*. (System.Security.Cryptography) | Microsoft Learn. Retrieved November 17, 2022, from <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.aes?view=net-6.0>

Dotnet-Bot. (n.d.). *Environment class (system)*. (System) | Microsoft Learn. Retrieved November 14, 2022, from <https://learn.microsoft.com/en-us/dotnet/api/system.environment?view=net-6.0>

Dotnet-Bot. (n.d.). *File class (system.io)*. (System.IO) | Microsoft Learn. Retrieved November 14, 2022, from <https://learn.microsoft.com/en-us/dotnet/api/system.io.file?view=net-6.0>

edurekaIN. (2021, February 26). *Amazon SES Tutorial | How to send emails using AWS SES | AWS training | edureka*. YouTube. Retrieved November 17, 2022, from <https://www.youtube.com/watch?v=gVRTKuMFc0c&t=1455s>

Furmanek, A. (2016, March 19). *Executing C# code using PowerShell script – random it utensils*. Random IT Utensils. Retrieved November 17, 2022, from https://blog.adamfurmanek.pl/2016/03/19/executing-c-code-using-powershell-script/

Hicks, J. (2014, December 16). *Add pop-up tips for PowerShell WinForms script: Petri it knowledgebase*. Petri IT Knowledgebase Add PopUp Tips for PowerShell WinForms Script Comments. Retrieved November 17, 2022, from https://petri.com/add-popup-tips-powershell-winforms-script/

KL, A. (2022, September 11). *Step-by-step procedure to install openssl on the windows platform*. The Sec Master. Retrieved November 17, 2022, from <https://thesecmaster.com/procedure-to-install-openssl-on-the-windows-platform/>

Lastname, O. (2010). Online journal using DOI (digital object identifier). *Main Online Journal Name*, Vol#(Issue#), 159-192. <https://doi.org/10.1000/182>

*Make email easier.* Thunderbird. (n.d.). Retrieved November 17, 2022, from <https://www.thunderbird.net/en-US/>

Sdwheeler. (n.d.). *Send-mailmessage (microsoft.powershell.utility) - powershell*. (Microsoft.PowerShell.Utility) - PowerShell | Microsoft Learn. Retrieved November 17, 2022, from <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/send-mailmessage?view=powershell-7.3>

Steven Rombauts. (2018, December 11). *Test SMTP with telnet or openssl • Steven Rombauts*. Steven Rombauts. Retrieved November 17, 2022, from <https://www.stevenrombauts.be/2018/12/test-smtp-with-telnet-or-openssl/>

Studebaker, N. (2017). *Fileless Malware Demystified*. Youtube. Retrieved August 27,

2022, from<https://youtu.be/atL1WmmMJJw?t=1589>