

Daemon - A Contest Revealed

Winner Announcement and Full Tutorial

Thanks to all who participated in Daemon: A Contest. Before we get to the winners as well as the tutorial on how to solve the challenge, EH-Net would like to once again thank Daemon author, Daniel Suarez, and all those involved in making this contest happen. It's amazing how a few crazy ideas can all come together into something fun and educational while at the same time spreading the word of this truly unique work of fiction.

What started as a little game to hide a secret message turned into another classic teaching vehicle for EH-Net readers. The image is a twist on the usual steganographic content. Øyvind Østlund and Adam Wardon crafted some C# source code to hide data in an image of the author which is also invisible to the Daemon's bots. What's in the message still is up to you to find, but three talented people found the message and took the action it recommended. Because of that, EH-Net members jason, blackazarro and ozpj have won signed, pre-release copies of Daemon, Hard Cover Edition. And now, with the coding expertise of regular EH-Net contributor, Ryan Linn, we will show you how it can be done using a couple tutorial files and all free tools.

del.icio.us

Discuss in Forums {mos_smf_discuss:News Items and General Discussion About EH-Net}

The challenge starts simply with a referral to a webpage, the About the Author page on the official Daemon site. When you get to the webpage, you will find a strange looking picture of Daniel Suarez. The image is made up of colored text put together to form a graphical representation of Mr. Suarez. This colored text is actually our hidden challenge.

Or as Daemon's author puts it, "In typical steganography implementations, users would conceal textual information in the bitstream of a JPEG or other graphical format. But here, the very presence of a photo is hidden from web bots and spiders (i.e., the author's photo is undetectable to bots). Likewise, the message hidden within the hidden photo is encrypted -- so there are actually two messages in the same stream; one graphical (but invisible to bots and thus unsearchable) and the other textual (and invisible to all but skilled practitioners)."

By copying all of the text from the image above and pasting into Windows Notepad, you can view all of the text to read it better.

The first few lines have some good information in them. Let's simply start by analyzing just the first few sections.

The first part of the text starts with `//` which denotes a comment. Comments are sections of the code not processed by the compiler and are used by developers, amongst other reasons, to leave notes for others who might never get to talk to the original authors of the code. Let's take a closer look:

In this case, they left us with `SetRegexFlagToSearchWhenDateChangedFromValue`, but at this point it's not initially clear why. Even if we have no clue what they are trying to tell us, we can make note of interesting words such as `search when date changed`. We will skip over it for now.

It initially appears that the text is separated by a tilde (~) character. A quick glance down the line, we see it appears between other pieces of data as well. The next set of text is clearly a date, and it isn't immediately clear what that is for either. It does seem to make sense that it would go with the previous field that references a date, but we still don't have enough information to make a move, so on we go.

The next set of text is `DESCryptoServiceProvider` which seems to have some potential. If you head to Google on this term, you will find a Microsoft reference for this function, and, along with a reference to C# on the author's about page, we can infer that this code is probably DES encrypted using Microsoft C#.

One of the Google search results returned is the Microsoft Developer Network (MSDN) Reference Site. By clicking on the `DESCryptoServiceProvider Members` link on the left hand side of the MSDN site, we can see what methods the `DESCryptoServiceProvider` class has. Looking through the listed members, a few fields are visible as properties. These include `blocksize`, `feedbacksize`, `IV` and `key`.

The `IV` and `key` we see represented in the text that we just found on the webpage. Cool. Looks like we're on the right track. The data from the webpage references `UseGragg'sKey`. A Google search on `Gragg Daemon` finds some book reviews which reference Gragg as a main character in the book `Daemon`. Going through the book, we locate Gragg's key and write it down for later use. Oh we're sorry, but we're not going to tell you what it is...

did you think we'd make it that easy for you?

Moving forward in the information we retrieved from the website, we notice what appears to be a structure definition, "PrivateIV()AsByte={&H12,&H34,&H56,&H78,&H90,&HAB,&HCD,&HEF}." This seems to be the IV that is referenced in the DESCryptoServiceProvider class, so we will also save this piece of information. The rest of the text appears to be a series of messages, each one starting with "|-BEGIN-|" and ending with "|-END-|." Starting here, we will save the rest of the text to a file called "DaemonText.txt" on our desktop, getting rid of all the text we have before the first "|-BEGIN-|."

Before we start writing any code, it would be handy to know what type of data we have between the tags. Knowing that DES encryption's output wouldn't be all text, and noticing that the alphanumeric text appears to be padded with an equal sign, we can make an educated guess that the DES encrypted data is encoded as Base64, a standard encoding scheme that will allow our binary to be encoded as ASCII text.

We now have enough information to start writing code. If you don't have a Microsoft C# compiler, then go to <http://www.microsoft.com/express/vcsharp/> and download and install Visual C# Express Edition for free. The above link also has some great resources for beginning developers. If you think you may code in other Microsoft languages, feel free to download the entire Visual Studio Express Edition, but just the C# version will work for this tutorial. If you plan on doing any additional development after trying out this exercise, installing the database component may be beneficial as part of the install process. Once installed, make sure that all of your libraries are in place, and then you may want to restart your computer.

Start Visual C#, go to the File menu item, and choose "New Project" for the following dialog box to appear:

Choose "Console Application" and make the name "DaemonContest" and click "OK". You should now see a screen which looks something like this:

This is Visual C#'s integrated development environment (IDE) with the basic constructs of your new programming project created automatically for you in the largest area. The first two lines begin with the word "using."

The "using" keyword tells the C# language to include other libraries of methods, classes, and variables. These libraries contain the basic items we will need to build our application. We will later add more as needed. Next are the basic commands necessary to create our program.

The namespace command defines the scope or context of our application, which is the `DaemonContest` application. Next we define our class, called `Program`, which will hold the instructions for our application. Finally is the method `Main` which is defined as a static void method. The static designator means that this method will remain the same in all instances of our program class, which for our purposes today really doesn't matter much. The `void` means that nothing will be returned from our method. The method is named `Main` indicating that it will be the main block of code from which the application will start, and it will be passed an array of strings named `args` or the other arguments passed to the application on the command line.

Next in the application comes the definition of the variables that we have found. We specify the IV (initialization vector), our key that we found, and the delimiter for our messages, which I chose to use the ending flag of each message. The reason I chose to use the ending flag was because after looking through the data, I noticed that the last part of the data on the page didn't end with an end flag. It just ended. So if there is an end flag, then we know that the data is complete. Because this means that we may end up with some data that is incomplete, we need to add in some error checking later on.

The next thing that needs to be created is the portion of the program to handle the input. The application expects that the name of a file with the data that we have captured from the author's page to be used as the only argument to the program. The application checks the arguments passed to it via the `args` that were declared as part of the `Main` function. If there is less than 1 argument, then the program will instruct the user as to the proper usage of the program and then return, which in this case will cause the program to exit.

Don't worry if this all seems foreign to you. If it didn't, you probably wouldn't need to read this tutorial. Now is probably a good time to share the link to the sample Visual C# file named `DaemonSolution.cs` that you can use to follow along with this tutorial. Sorry for the delay, but there's actually a very good reason why it is done in this order. If you just open the sample C# file in Visual C#, you will not see the build menu item. That's why it's important for you to create a console application project first, then open the sample file. Keep your head up!

This next step is not required, but it is a good practice for proper user feedback. Before any work can be done on the file, we check to ensure that the file exists. In order to do this, the `FileInfo` object is used. A new `FileInfo` object is created named `f` using the first argument that is passed into the program, which is referenced by `args[0]`. Once the `FileInfo` object is created, the `Exists` method is used to check to see if the file exists, and, if it does not, a message is shown to the user indicating that the file wasn't found, and then exits. This is also a good way to identify typos.

Once the file is known to exist, the next step is to read the data from the file. To do this the first step is to create a `StreamReader` object named `s` using the `OpenText` function from the `FileInfo` class. At the same time the application initializes some strings that will be used to read the file into memory. The data string will hold all of the data that has been processed when the reading is finished, and the line variable, as you probably guessed, will hold each individual line.

The next set of code is a loop that used `ReadLine` from the `StreamReader` object to read in a line at a time, and then appends it to the data string that we will actually decrypt later. Once the lines have been read and appended to the data string, the strings need to be broken into messages. The `Split` function of the data object is used here to split apart the data string into the array called `parts` which contain each individual message. Because we are splitting based on the ending tag, it will not be part of the message, but when we go to decrypt the data we will need to remove the begin tag from each part before processing.

Next we need to setup our encryption system. The DESCryptoServiceProvider object will provide our decryption functionality. Because this object is not included by default in the libraries, we will have to add the System.Security.Cryptography library to the top of our application. Next we create a new object named crypto that we will need to setup in order to decrypt the data. Once it is created, we set the IV to the IV that we obtained from the author's page. Based on the data from the MSDN page, we know that the crypto key variable is a byte array. Seeing as how we have all read Daemon in its entirety, we now have the text for Gragg's Key, right? Because we currently have our key in string form IE text, we will need to convert the text to a byte stream. To do that, we use the System.Text.ASCIIEncoding object named 'encoding' to use the GetBytes function on our key. Then we assign it to the Key variable. Again, this library isn't included by default, so we must include the System.Text library up top in order to use this function. The crypto object should now be setup for decryption.

To do the encryption, I chose to create a separate function called 'Decrypt'. There is still a little bit more to be done in our main function, but for now let's go work on the DeCrypt function. We will have the DeCrypt function return our decrypted string, so we will define 'Decrypt' as a static function that will return a string when it is supplied a Base64 encoded string and a DESCryptoProvider object named crypto.

We will begin the DeCrypt function by creating our clean base64 string by taking our encoded data, and, if it exists, removing the begin tag from the string. Because we know that at least one message from the page isn't complete, there will be failures in decryption for at least one part. Because of the failure, we will wrap our decryption in the 'try' statement which will allow us to catch an error and display a message instead of the application crashing. Good idea, huh?

The decryption itself comes next. Because the decryption needs to work on a byte array instead of a base64 encoded string, we use the FromBase64String function to assign the decoded bytes to our byte array, encString. The function that will allow us to read back out the decrypted data requires that we pass the byte array in as a memory stream, so we create a MemoryStream object called 'ms' out of our encString byte array. We pass that memory stream along with our crypto object's CreateDecryptor return value to a CryptoStream object, and specify that we will be reading from the stream. Finally we create a StreamReader object 'sr' to read our decrypted data stream.

As we now have a way to read our decrypted data (if it didn't fail) and return an error message, we now create two string variables, var and message. These two string variables allow us to read one line at a time from our stream reader object and append it to the final message that we will return back to the calling function. Now we are done with our function, so we need to finish implementing the decryption in our main function.

Back in main, we need to create a loop to print out the decrypted value of each of the parts that we have split out of the author's page. For each part in our parts string array, we will write out the decrypted value of that message using the DeCrypt function and passing in the part of the message we are decrypting, along with our crypto object.

Now it's time to build our binary. First let's save our project by going under the File menu and choosing 'Save All.' Next go to the Build menu and choose 'Build Solution.' Now that the solution is built, we can take the data that we have saved from the author's page, starting at the first begin tag and going until the last piece of data and save it to a file in our project directory. Our project directory can be found under 'My Documents' in the 'Visual Studio 2008' directory. Navigate through 'Projects' to the

“bin” then “Release” directory of our computer. Once the file is saved, open up a command window by going to the start menu, choosing run and then typing in cmd.exe.

Once you have navigated to the Release directory in the cmd window where we saved our file, typing in a dir should show both the file that we saved as well as the DaemonContest.exe file that we built. Run the application against the file with our saved data by typing “DaemonChallenge.exe myfile.txt” where myfile.txt is the file that you saved. You should now see the output of the decrypted data.

Without giving away the secret, we can at least mention that the encrypted message is a special note from the author about a private blog that will be updated with new and exciting details of the world of Daemon and only seen by those with the skills to find it. He also has an action item for you, which is how we determined the winners. Finally he tells you the purpose of the datestamp. Since the blog may be regularly updated, he recommends the creation of a script that will automate the grabbing of the text and decrypting it. To do the grabbing and decrypting, Perl has a number of modules which will facilitate the decryption and grabbing of data, so that will be our language of choice. In a future tutorial, we may show you how to do this in Python, a language growing in popularity in security circles.

For the sake of making sure that everyone gets similar results, we need a working environment that includes everything we need to create our Perl script. Wouldn't it be nice if there was a Linux distro as a VMware image that fits the bill? As luck may have, BackTrack 3 (BT3) works perfectly, and most of you have it already. In order for the script to work you will need some type of network connectivity. We recommend using the BT3 VMware Image preferably through VMware Player with either Bridged Mode or NAT.

Once you have BT3 booted, open up your favorite editor and create a file called DaemonSolution.pl. If you don't have a favorite editor, SciTE (included in BT3) has language aware highlighting that will help tip you off if you make any typos while working on a script within BT3.

As with the C# portion of this tutorial, we recommend that you download the sample perl file directly from within BT3 using the link in this article. Download DaemonSolution.txt and rename to DaemonSolution.pl to follow along with the code during the discussion of each section.

The first section of the file lays out which interpreter that we will use, so that the operating system will launch the Perl interpreter when the application is launched. In BT3 Perl is located in /usr/bin. If you are not using BackTrack it may be in another location. After we define the interpreter, we add the header to the application so that if someone casually checks the application they will know what it is and what it is for. After that we define what libraries that we will use. The Crypt::CBC library is going to facilitate our decryption. It will allow us to use CBC or (Cipher Block Chaining) to decrypt arbitrarily long data segments, otherwise you will be limited to 8 blocks of data without it.

The MIME::Base64 module will allow us to decode the binary data that is DES encoded so that Crypt::CBC can help us decrypt it. Finally LWP::UserAgent will include the Perl LWP modules that we will use to get data off of the web from the author's page for our application to decrypt. Once these modules are “use”d or included, then we will have the additional functionality we need that isn't in base Perl to get the web page, find our data, parse the messages, and

decrypt the data.

The next set of lines defines variables that will be used in comparing the contents of the blog to check for new updates including the file that contains our last retrieved timestamp. This way we can know if there have been any updates since we last checked. The next line contains the URL of the author's about page, followed by an arbitrary date that we would want to compare the changed date against and finally the key which we found in the book.

With all of the variables that we may need to change if the author updates something set in place and our libraries included, we are now ready to start the code to do the work in our application. The first thing we will do is create a request object that will call the HTTP `GET` command on the URL that we specified as the author's about page. Next we create a UserAgent object and have the UserAgent object request the webpage using the request object. The response object, which we called `$res` will be returned from the UserAgent. We check to make sure that the response indicated a successful return of the webpage, and if so we assign it to the variable called `$data`. If not, we print out a helpful error message indicating that it failed, and exit the program gracefully.

Now that we have the HTML source from the webpage in the variable called `$data`, we have to find where the author's embedded code is in the HTML source. By viewing source on the author's about page in your favorite web browser, and searching for the string `stegoimage`; you can see that the entire encrypted message is surrounded by a div tag named `stegoimage`. Knowing that all of our data is wrapped in a div tag, the easiest way to get at the data in Perl is going to be to write a regular expression (regex) in order to capture our data. The regex we create searches the `$data` string for the first occurrence of the word `stegoimage` followed by the ending of the div tag. We simply take everything between the two div tags. Our regex uses the `/s` specifier in order to ignore newline characters, otherwise we would only get part of the data we needed. We assign the text that we found to the variable named `$string`, so that we can parse only that information without dealing with the rest of the HTML.

Once we have our `$string` variable, there is still a lot of HTML in the string that we need to eliminate, before we have the raw data. Each character in the string is wrapped in the HTML required to colorize it in order to make the picture of the author. To get rid of it, we will remove anything wrapped in `<` or `>` characters of the HTML. All that is left is the text that makes up our secret data. There are also new lines left in the data so the next line of code removes all of the spaces and new line characters which may be left in our string. Our string should now just contain the hidden message from the page without any of the HTML and all combined into one single line without any breaks or spaces. Next comes the string parsing in order to get all of the goodies that we will need to decrypt the message in case things change.

The first thing that we had in our string was the information about when the last time the blog was updated. Looking at our data from before, we know that the data looked like a `~` character followed by the data delimited by `/` characters. We write a regex to look for that format of the date, and, if we find it, we save it to the variable named `$date`. If we don't find it, then something is wrong with our parsing that has happened thus far. If this is the case, then we need to stop in order to present the user a message indicating that something is amiss and that we need to exit.

We know from the first program that our IV is kept in a string surrounded by `{}` and `}` characters. That should be the first thing in our string, so we can create a regex to find our IV. The next set of code finds our IV and assigns it to the `$v` variable. If it is not found, we again practice good coding techniques by printing an error message indicating that we can't continue without it and exit the program.

We specified our timestamp file up at the top of the application, and it is now time to open the file and check to see if the date has changed in the string on the webpage. We check to make sure that the file exists with the `&f` operator and, if it does, we open the file, read in the last time it was modified into the `$time` variable and close our file. In case there were any spaces, we use the `chomp` function of Perl to remove any trailing spaces or newlines. If the file did not exist, we had pre-populated the `$time` variable with an invalid date, so they will not match, and we are guaranteed to see any updates.

The next set of code compares the two dates, and if they are equal then we print a message indicating that there are not any changes and exit our program. If the dates don't match, then it is time to do our decrypting. Before we can decrypt the data we have to parse it into the individual messages. To do this, we will start out by getting rid of everything up until the first `|` character where our messages start. We do this by using the `regex` replacing everything at the start of the string up to and including the first `|` character with just the `|` character, so that our first begin tag is intact.

Next we take our string and search for all of our messages which are complete with both a beginning and an ending tag, and, if they are complete, it will put them into an array called `@strings`. This statement is different from all of the other `regexes` we have used in that each of the others that searched for something would only yield one result back, but this `regex` will return all of our messages that are found.

We found our IV earlier and assigned it to `$v`, but we need to do more parsing in order to get it into an actual byte vector from a string. The first thing that we will do is remove all the instances of `&H` from the string. After that we should have a string containing hex values separated by commas. Now we need to turn each of these hex values into the binary character that they correspond to. We do this by splitting our string of comma separated values into an array which we call `@a`. Then we go through these characters one letter at a time in our `foreach` loop. The `foreach` loops gets the character value of the hex character represented by the hex value in our string and appending it to the string named `$v`. By the end of our loop, we have our key in binary stored in the `$v` variable.

It is now time to create our decryptor object. We create a `Crypt::CBC` object using the key that we found in the book, `$v` as our IV, specify that our key is the literal key and indicate that there is no header on each message we will be decrypting. We call our new decryptor object `cipher` and then get ready to do our decrypting.

The next block of code does all of the crypto work. We create a variable named `$i` that we will use to control our loop and have it initially assigned the value of 0 which will be the index of the first part of our strings array. We will keep looping and incrementing `$i`, until we are all the way through the last message. Before each message we will print a block of `-` characters all on the same line to indicate a separate message in the blog. The next line in the loop says to use the `decrypt` function of the cipher object to decrypt the value returned by the output of the `decode_base64` function. We pass the current message in our `@strings` array to the `decode_base64` function, then we follow our decrypted data with an additional newline in order to enhance readability.

Almost there. We print out the timestamp into our timestamp file, and we are now done. We save our file to our home directory, `/root`. Next we exit our editor and open up a terminal window, the icon that is directly beside the `K` icon in the bottom left of the screen. Once we see our prompt we will type in `chmod 755`

DaemonSolution.pl” and hit enter, making our program executable. Finally it's time to read the blog. Whew! If you type in ./DaemonSolution.pl and hit enter, you should see the blog.

If not, it could possibly be the sample Perl file. Opening this file from within Windows first and then moving it to the BT3 virtual machine could cause the following compatibility error: -bash: ./DaemonSolution.pl: /usr/bin/perl^M: bad interpreter: No such file or directory. To fix this, simply type “dos2unix DaemonSolution.pl” while still in the same directory. You should also chmod again to make sure that the file level permissions are correct, as dos2unix has a tendency to reset them. If it didn't work earlier, it should now.

One final step. Checking to make sure that the date comparison works, we can run the application one more time to ensure that it prints out our message indicating that nothing has changed. If all of that works, then we're done!

While not a complete step-by-step explanation, this should give you plenty to understand how it is done and give you a little incentive to go out there and fill in the gaps on your own. After all, that's what hacker's do, right?

Happy Holidays,

Ryan & Don of EH-Net