SANS | GIAC CERTIFICATIONS

WHITE PAPER

Base64 Can Get You Pwned

Kevin Fiscus

Base64 Can Get You Pwned

GIAC (GCIA) Gold Certification

Author: Kevin Fiscus, kevinfiscus@gmail.com
Advisor: David Shinberg

Accepted: April 13th 2011

Abstract

Base64 is an encoding scheme originally designed to allow binary data to be represented as ASCII text. Widespread in its use, base64 seems to provide a level of security by making sensitive information difficult to decipher. In reality, the use of base64 provides a significant advantage to attackers while providing minimal benefit to defenders. The use of base64 can result in the disclosure of passwords, bypass of data leakage protection systems and can even be used to create a one click, obfuscated and self-contained cross site scripting attacks. Because of these risks, detecting base64 usage on a network should be an important part of any comprehensive security program. Unfortunately, there is a problem; base64 is almost impossible to detect accurately using traditional methods. This paper provides an overview of the base64 problem, and more importantly, outlines a methodology that can be used to promote base64 detection using the Snort intrusion detection system.

1. Introduction

Helix Pharmaceuticals is worried about security. In the cutthroat world of multi-billion dollar pharmaceutical companies, industrial espionage is a significant concern. In addition, political and social activists continually attempt to disrupt business as retribution for perceived injustices. As a result, Helix takes information security extremely seriously. Their security program consists of numerous protective and detective controls including the use of extremely strong passwords, data leakage protection (DLP) solutions, network intrusion detection systems (NIDS), web filtering and email security solutions. The controls in place were deemed, by the Chief Security Officer, to be adequate until they discovered that their strong passwords were compromised, their DLP and IDS were evaded and their web security controls were bypassed. After a thorough investigation, it was determined that one simple technology was the cause of it all – base64. This story is fictional but the concepts are real and deserve the attention of every information security department.

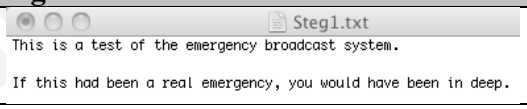
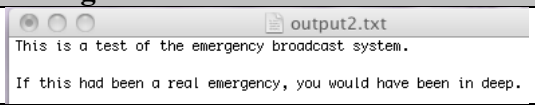
Base64 is a commonly used encoding scheme originally designed as a way to represent binary data in an ASCII text format. Like almost every aspect of computer technology today, base64 if not used properly, can result is increased security risk. As mentioned in the story about Helix, attackers can also use it as a method to obfuscate and/or execute their attacks, evade detection and to bypass otherwise strong controls. To mitigate the risks associated with use of base64, it is important to understand what base64 is, how it is used, how it is abused and how to detect its use in modern computing environments.

2. Base64 Overview

2.1. Encoding vs. Encryption

When it comes to obscuring data, there are really three different approaches commonly discussed: steganography, encryption and encoding. Steganography, or "stego", is a process by which data is hidden from observers. Herodotus documented one of the earliest examples around 440 BC. He tells the story of Histiaeus who shaved the head of his most trusted slave and tattooed a message in it. Once the slave's hair had grown back, the message was hidden. (Perera, 2011) When the messenger got to their final destination, their head would be shaved thereby disclosing the message. In today's modern age of computing, a similar effect is achieved by changing the least significant bits of each byte of an image file, for example. In pure steganography, the data is not changed in any way, but is simply hidden.

The following two pictures look similar. The one on the left is the original. The one on the right has had data injected into it using a program called iSteg. To the naked eye, there are few, if any, visible differences between these pictures, however if the second picture were fed into the iSteg program, the original text would be revealed.

Original Picture	**Stego'd Picture**
	
Original Text	**Un-Stego'd Text**
 Steg1.txt This is a test of the emergency broadcast system. If this had been a real emergency, you would have been in deep.	 output2.txt This is a test of the emergency broadcast system. If this had been a real emergency, you would have been in deep.

Encryption is an entirely different method of obfuscation but rather than hiding the fact that a message exists, like stego, encryption attempts to hide the meaning of the message. One of the simplest forms of encryption is a rotational cipher where the letters of the alphabet are shifted. A rotation of 3 or ROT-3 would result in two alphabets, the true alphabet, in which the original message is written and the shifted alphabet. The following shows a typical ROT-3 scheme.

True: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Shifted: CDEFGHIJKLMNOPQRSTUVWXYZAB

Using this ROT-3 scheme, the letter C would be used in place of A so the word CAR would be encrypted as ECT and the word HOUSE would be encrypted as KQWUG. This is, of course, a very basic encryption scheme. Modern cryptographic schemes use sophisticated combinations of substitution and transposition against blocks or streams of data to come up with ciphertext that is difficult, if not impossible to convert back to the original plaintext without the proper key. Encryption is an effective way to protect the confidentiality of data.

The following table shows the same text encrypted using the same encryption scheme but using different keys. Encrypting data results in a binary, rather than a text file thus the binary results have been encoded using base64 to make the readable.

Clear Text	Algorithm	Key	Base64 Cipher Text
Hello!	DES	Test	OBfxMpyn7oY=
		Test1	5Rcw8GZ+/QM=
		TestTest	q2a0ZkvgMeM=
		test	uy8XtiCOto0=

As you can see, other than the trailing equal sign (a result of base64 padding), different keys used to encrypt the same source text using the same algorithm result in vastly different encrypted or cipher texts. Decrypting the cipher text without the key is ranges from difficult to virtually impossible depending on the strength of the encryption algorithm.

Encoding may seem like encryption in that data gets changed from one form to another and the encoded text does not look like the original. Encoding, however, does not use substitution and transposition based on a secret key. Rather, encoding is the process of displaying data in another format. In the world of computers, the most common form of display suitable for humans to read is the American Standard Code of Information Interchange or ASCII. ASCII includes the letters and numbers we read every day plus some control characters such as backspace and tab. Thus all of the letters, spaces and punctuation written in this document so far are representations of ASCII text.

In the world of computers however, ASCII is not the only way of encoding or representing data. In its most basic form, a single ASCII character is stored on the computer as a single byte of data that can also be represented as binary, octal, decimal or hexadecimal. The following table shows the various encodings of some common ASCII characters:

Glyph	Hex	Dec	Oct	Binary
A	0x41	65	101	100 0001
a	0x61	97	141	110 0001
!	0x21	33	041	010 0001
Backspace	0x08	8	010	000 1000

Based on this, a simple word like Cat can be represented as follows:

- ASCII: Cat
- Hexadecimal: 0x43 61 74
- Decimal: 67 97 116
- Octal: 103 141 164
- Binary: 01000011 01100001 01110100

All of these encodings spell Cat and as long as a recipient knows enough to decode the message, they can. The fact that the message may be encoded provides no assurance of confidentiality other than relying on the fact that any given attacker may not be able to determine the method of encoding. Unfortunately, as you can see from the example above, many types of encoding often used in the computing industry are fairly easy to identify.

Like ASCII, hex, octal and binary, base64 is an encoding scheme. Specifically, base64 was designed as a means to represent binary data as ASCII text using a numbering system consisting of 64 digits. This may seem difficult to understand, but it is fairly simple. We typically interact with numbering systems with 10 digits; 0 through 9. This is a base10 system. Binary, a base2 numbering system, has 2 digits; 0 and 1. Hexadecimal is a base16 system using 0 through 9 plus a, b, c, d, e and f for its digits. Base64 typically uses 0 through 9, a through z and A through Z for the first 62 digits of the system. Different variations of base64 use different characters for the final 2 digits.

Just as ASCII and binary can be used to represent data, so can base64. The palindrome "Was it a car or a cat I saw" would be represented as "V2FzIGl0IGEgY2FyIG9yIGEgY2F0IEkgc2F3". As you can see, the source phrase reads the same forwards as it does backwards but this is not the case in the encoded text. While this may seem "secure" the fact that you can simply paste this text into an online base64 decoder and recover the original text illustrates the weaknesses of base64 as a security mechanism.

2.2. Common Use

Base64 is used virtually everywhere. The following are some common applications that make use of base64.

- Basic authentication to web sites. When this type of authentication is used, the username and the password are separated by a colon, concatenated and the results encoded using base64. (Franks, 1999)
- Transfer of binary data via mediums such an email, as a replacement for uuencode. (Freed, 1996)
- Evasion of basic anti-spamming tools. (Craig, 2007)
- Encoding characters strings in LDAP LDIF or files (Good, 2000)
- Embedding binary data in an XML file
- Encoding binary files, such as images, within scripts or HTML to avoid depending on external files. (Coyier, 2010)
- Communicating encrypted cookie information (Prabhakar, 2011)

Of these uses, only a few should be considered both legitimate and appropriate. Using basic web authentication, for example, should be avoided as it risks disclosing the username and password to an attacker. Malicious use of base64 to evade anti-spam technologies is obviously not recommended. The remaining use cases are but should be considered suspect for a variety of reasons that will be discussed in detail throughout this document.

2.3. Identification and Decoding

The characteristics that make up a base64 encoded string are fairly simple; it will typically contain letters (A-Z and a-z), numbers (0-9) and the characters "/", "+" and "=" where the equal sign, if found, will always be found at the end of the string. Base64 strings usually contain a multiple of 4 characters (e.g. 4, 8, 12, 16, etc.). In such cases, the minimum size for a base64-encoded string is 4 characters. If the source string is not long enough to generate an output of 4 characters, one or two equal signs will be added for padding. This padding is found in most base64 encoded strings where the encoding does not generate a number of characters that is divisible by 4, thus you often see either one or two equal signs at the end of base64 encoded data. Based on this definition however, the words "data", "Data" and "Database" are all potentially valid base64 (although they decode to random binary data) making positive validation of base64 data difficult. Making things worse, base64 does not always use the special characters / and +. In some implementations of base64 a number of other special characters are used including the dash (-), the underscore (_), the period (.), the colon (:), and the exclamation point (!). In addition, some implementations of base64 don't use padding. As a result, base64 can contain any combination of letters (upper and lower case), numbers and various special characters (/+-_:!) that may or may not have one or two equal signs at the end. Needless to say, detecting base64 in your organization can be difficult.

There are a number of methods to determine whether a specific set of data is a valid base64 encoded string, but determining whether it was actually the result of base64-encoding is virtually impossible by any means other than trying to decode it. Fortunately, in many cases, detecting base64 encoding is not really desirable as such encoding has numerous legitimate uses. What we are often concerned about is the use of base64 to "secure" authentication credentials and that can be detected using, for example, Snort as seen in the following Emerging Threats rule:

```
alert tcp $HOME_NET any -> any $HTTP_PORTS (msg:"ET POLICY Outgoing
Basic Auth Base64 HTTP Password detected unencrypted";
flow:established,to_server; content:"|0d 0a|Authorization|3a 20|Basic"; nocase;
content:!"YW5vbnltb3VzOg=="; within:32; classtype:policy-violation;
reference:url,doc.emergingthreats.net/bin/view/Main/2006380;
reference:url,www.emergingthreats.net/cgi-
bin/cvsweb.cgi/sigs/POLICY/POLICY_Basic_HTTP_Auth; sid:2006380;
rev:10;)
```

This rule is fairly straightforward, particularly when you remove the messages, ID numbers, references and revision information as follows:

```
alert tcp $HOME_NET any -> any $HTTP_PORTS (flow:established,to_server;
content:"|0d 0a|Authorization|3a 20|Basic"; nocase;
content:!"YW5vbnltb3VzOg=="; within:32;)
```

This rule is looking at TCP traffic on $HTTP_PORTS (a variable used to define the ports on which web traffic is expected) for specific content. In this case, it is looking for bytecode (hex representation of binary data) of "0d 0a", the word "Authorization", bytecode of "3a 20" and the word "Basic". None of the above is case sensitive. Adding further specificity, any communications with "YW5vbnltb3VzOg==" found within 32 byes of the previous match would be excluded. (The string starting with YW5 is base64 encoding for "anonymous:". This approach identifies "basic web authentication", one of the most common uses for base64 and one that almost always involves usernames and passwords.

Detecting basic web authentication may be interesting but it is not always sufficient. User credentials are not the only pieces of sensitive information that can be encoded using base64. Consider the pharmaceutical company that deployed a complex Data Leakage Protection solution in an effort to protect their newest multi-billion dollar drug. Their DLP solution is configured to watch for a specific string of characters; "super secret formula X+3(Y)/437*Q". An insider seeking to bypass that system could simply send it out as "c3VwZXIgc2VjcmV0IGZvcm11bGEgWCszKFkpLzQzNypR" which is the base64 encoded version of that same formula. Unless the DLP solution has been configured to look for the base64 encoded string, it will be missed.

As discussed previously, determining that a given data string is actually base64 is not possible without attempting to decode it. That said, identifying strings that are consistent with base64 encoding can be done using Perl Compatible Regular Expressions. This must be done carefully as this approach is subject to significant false positive or false negative results. For example, a regular expression "[0-9a-zA-Z+/=]{20,}" could be used as it looks for a string of characters that is at least 20 characters long containing letters, numbers or the special characters listed. When analyzing typical human-readable text, this approach may be reasonable as 20 character words are uncommon, however a long URL such as http://www.something.com/something/somethingelse/somethingmore, would result in a positive match to the regex. Another problem with this approach is that it only looks for encoded text of 20 characters or more. This would fail to detect an encoded password (for example) that is as long as 12 characters. While this approach has a role in an overall base64 detection scheme, because of its weakness, another, more specific approach is necessary.

The following regular expression is more complex but does a more comprehensive job of identifying base64

- (?:[A-Za-z0-9+/]{4}){2,}(?:[A-Za-z0-9+/]{2}[AEIMQUYcgkosw048]=|[A-Za-z0-9+/][AQgw]==)"

This can be more easily understood by breaking it down into its individual parts. Basically it is looking for two groups of data as identified by the two sets of beginning and ending parenthesis. The first group, (?:[A-Za-z0-9+/]{4}){2,}, looks for two or more

groups of 4 characters that match the listed letters, numbers or special characters. *Note: the "?:" is used to optimize the processing of the regex and doesn't affect what the regex is looking for.* The second group looks for either:

- Two characters matching A-Z, a-z, 0-9, + or / followed by one character (AEIMQUYcgkosw048), followed by an equal sign.

or

- One character matching A-Z, a-z, 0-9, + or / followed by an A, Q, g or W followed by two equal signs.

The result will be at least a 12-character string meaning the source data was at least 7 bytes in length. This approach results in very few false positives however does result in significant number of false negative results, or missed base64. This is because not all base64-encoded data ends with either one or two equal signs. An equal sign only occurs in some implementations of base64 encoding and is used to pad the data to ensure output is in four bytes blocks. Specifically, source data that has a multiple of three bytes of data (e.g. 3, 6, 9, 12, etc.) would result in base64 encoded data with no equal signs and would be missed by this regular expression. This also assumes the specific implementation of base64 actually uses padding. Also, there is no absolute standard for base64 ASCII character usage. All implementations of base64 use the characters 0 – 9, A – Z and a – z but that only addresses the requirements for 62 of 64 necessary characters. Most implementations of base64 use the forward slash (/) and the plus (+) however this creates problems in certain circumstances. For example, if base64 were to be embedded in a URL, the use of the forward slash would be interpreted as a URL divider rather than part of the base64. As a result, other characters such as dash (-), underscore (_), period (.), colon (:) and exclamation point (!) are used in some implementations.

The concerns related to the use of different special characters are fairly easy to resolve using additional regular expressions in which other characters replace the slash and plus. Unfortunately, the problem associated with the missing equal sign is far more difficult. Modifying the regular expression to not require any equal signs creates a large number of false positive results and is thus virtually useless. As a result, we are left with

an undesirable option: we either generate false positives or we generate false negatives. The best approach depends on the business problem you are trying to solve.

3. Understanding the Problem

Understanding base64 and how it can be identified is interesting as an intellectual exercise. To be meaningful in a practical sense, it is also important to understand why base64 represents a problem. The use of base64 places businesses and other organizations at risk in a variety of ways. Base64 can be use to compromise environments passively, with attackers sniffing network traffic to identify sensitive information including usernames and passwords. Base64 can be used actively to bypass data leakage protection or other data-focused security controls. Based64 can even be used to directly attack many endpoints. This conbination of threats makes it both difficult to detect and significantly damaging to even well protected organizations.

3.1. Password Disclosure

Password disclosure may be the most obvious risk associated with base64. Consider the fictional pharmaceutical company discussed earlier. They require users to select complex passwords of at least 14 characters in length and require that they be changed every 30 days. Using the most sophisticated computing methods available, brute force cracking a 12-character password consisting of only lower case letters would take approximately 3 years (assuming the cracking environment can guess 1 billion passwords per second. Cracking a 15-character password consisting of only lower case letters using the same computing enviornment would take over 53,000 years. (Password Recovery Speeds, 2009)

Brute force cracking, however, isn't always necessary. If the organization, out of ignorance for example, uses basic web authentication, or if the user uses their corporate password for a third party application that uses basic web authentication, the password can be disclosed by sniffing traffic on a local coffee shop or fast food resturant's wireless network. This is because the username and password in basic web authentication are encoded using base64, then passed to the server. There is no encryption involved.

In addition, some "behind the scenes" applications, such as anti-virus solutions, use base64 to encode the authentication controls between the client and the signature update server allowing an attacker to "steal" licenses. Specifically, when testing the effectveness of the Snort rules definded throughout this document, it was discovered that basic web authentication was used by a major anti-virus vendor to allow anti-virus clients to authenticate to signature update servers. The base64 used in the basic web authentication was able to be decoded revealing both the user name and password. An attacker could also use this fact to identify signature updates, conduct a man-in-the-middle attack and provide malware to the target masquerading as the update.

3.2. Data Leakage Protection Bypass

Many organizations today use some type of data leakage protection or DLP solution. These come in many forms ranging from those that are specific to one protocol (e.g. email) to those that "sniff" all network traffic. In virtually all cases, these technologies look for specific patterns of data such as an account number, a social security number or specific key words associated with other types of sensitive data. The use of base64 encoding can make this type of detection far more difficult.

Consider the relatively simple example of a social security number or SSN. An SSN is a 9-digit number that is often represented in the format of 123-45-6789 but can also be represented as "123456789", "123 45 6789" or a variety of other formats. Detecting SSNs effectively takes a fairly complex regular expression - ^(?!000)([0-6]\d{2}|7([0-6]\d|7[012]))([-]?)(?!00)\d\d\3(?!0000)\d{4}$. Unfortunately, after putting all of that work into the regex, an attacker can simply encode the SSN using base64 and wind up with MTIzLTQ1LTY3ODk=. The following table lists various SSNs encoded via base64.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
M	T	I	z	L	T	Q	1	L	T	Y	3	O	D	k	=
M	T	E	x	L	T	E	x	L	T	E	x	M	T	E	=
M	j	I	y	L	T	I	y	L	T	I	y	M	j	I	=
M	z	M	z	L	T	M	z	L	T	M	z	M	z	M	=

As you can see, there are some commonalities that could be leveraged to create additional regular expressions to detect base64 encoded SSNs, but the use of base64 makes detection far more difficult. In the above example, the commonalities are the result of using a specific implementation of base64 encoding and SSNs in the ###-##-#### format. Using different formats, different encoding schemes or even adding some number of leading characters (e.g. spaces, periods, dashes, etc.) adds complexity, and this is only one example of the type of data a DLP solution looks for.

If the word "Secret" is encoded using base64, the result is U2VjcmV0. Adding trailing information to the source data ("Secret 123") results in U2VjcmV0ICAxMjM=. As you can see, the first 8 characters are the same. If you add even a single leading space however, you get an encoded result, IFNlY3JldA==, that is significantly different. The same dramatic effect occurs when you make other fairly trivial changes to the source data as shown in the following table:

Source	**Base64 Encoded**
Secret	U2VjcmV0
Secret (1 leading space)	IFNlY3JldA==
Secret (2 leading spaces)	ICBTZWNyZXQ=
Secret (3 leading spaces)	ICAgU2VjcmV0
SECRET	U0VDUkVU
S E C R E T	UyBFIEMgUiBFIFQ=

These dramatic variations in output make configuring a DLP system to detect specific sensitive information extremely difficult and the complexities increase as the complexity of the sensitive data increases. While it may be possible to identify, and thus detect, the majority of possible combinations for a 6 digit word or even for something with a standard format, as a social security number, it is virtually impossible to do so for complex intellectual property or business data.

3.3. End User Compromise

There are numerous ways that an end user can be compromised using base64 that primarily rely encoding to evade malware detection signatures, IDS systems and similar controls. The best examples of such an attack involve targeting an end user via their web browser.

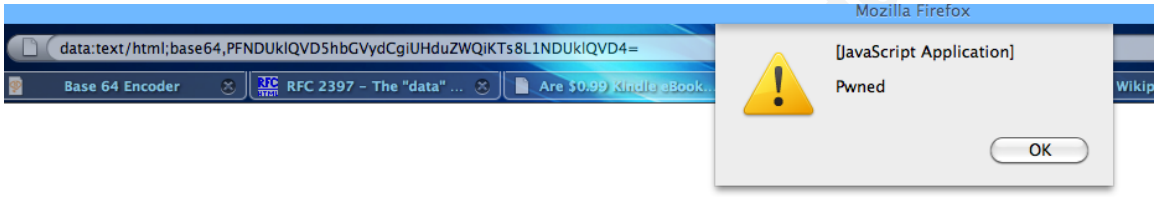
Web browsers are interesting in that they do a lot of the "thinking" for us. Originally designed to display ASCII text according to a set of rules called HyperText Markup Language or HTML, the functionality of web browsers has expanded significantly. One of the functions that most web browsers will do automatically is decode encoded data. ASCII text can be encoded in hexadecimal (base16), decimal (base10) and, of course, base64. This allows an attacker to embed malicious content such as JavaScript in a web site or a URL. Because the JavaScript is decoded by the browser, the actual JavaScript is not transmitted across the "wire" and thus is likely not going to be detected by IDS or other controls.

Consider a simple JavaScript "attack" - <SCRIPT>alert("Pwned");</SCRIPT>. Detecting this type of script is easy using a typical IDS, however it can be encoded using base64 resulting in - PFNDUklQVD5hbGVydCgiUHduZWQiKTs8L1NDUklQVD4= making detection far more difficult. This approach can be exploited by creating a very simple web page:

```
<html>
<body>

<h1>Heading</h1>

<p>Paragraph.</p>

<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;base64,
PFNDUklQVD5hbGVydCgiUHduZWQiKTs8L1NDUklQVD4=">

</body>
</html>
```

A user visiting this web page would see an alert box with the word “Pwned” pop up in their browser, but the JavaScript will have never been sent across the network, thereby evading network based detection.

This same approach can be used by pasting a link directly in a web browser’s URL entry field; specifically, the text “data:text/html;base64, PFNDUklQVD5hbGVydCgiUHduZWQiKTs8L1NDUklQVD4=” (without the quotes) will result in the JavaScript executing in a web browser as shown in the following image.



Getting a user to click on such an unusual URL is also not particularly difficult. The data URL scheme (as it is known) can be appended to a legitimate looking URL however there is an easier method – simply use a URL shortener such as TinyURL (http://www.tinyurl.com). Shrinking the text using TinyURL results in http://tinyurl.com/6bddyun. Given that users are familiar with compressed URLs associated with Twitter and Facebook, it is likely that they would not give such a URL a second thought. While this same attack vector could be used with JavaScript directly, sending JavaScript across the wire could be detected by an IDS or similar control while sending base64 would be less likely to be seen or blocked. Furthermore, using a data URL can allow the attacker to bypass certain protective controls. Specifically, the NoScript Firefox extension (http://noscript.net) is designed to block the execution of scripts, however presenting the script as a data URL bypasses this control resulting in the execution of a script in a browser that should block that type of activity.

This use of base64 to evade attack is particularly concerning. Using the combination of the data URL scheme, base64, JavaScript and URL shorteners, it is trivially easy to execute arbitrary code on a victim’s computer. The code would execute under the context of the web browser but this still provides the attacker with significant latitude in terms of attack options including the ability to establish an outbound, SSL encrypted communications channel. As most organizations have stateful inspection

firewalls, the response traffic to an established outbound session is allowed thereby allowing the attacker to bypass many different types of perimeter controls. This type of attack is not, in any way, sophisticated or difficult requiring only a basic understanding of JavaScript, access to a base64 encoder and access to a URL shortener.

The attack, however, is limited by the fact that it won't work in some web browsers. Modern versions of Internet Explorer do not decode most Base64 and while Google Chrome will, it will not execute the 302 redirect from TinyURL. Google Chrome will decode the base64 and will execute the resulting JavaScript, thus simply hiding the data URL information behind "Click Here" or similar innocuous text would likely be successful. Many of the web browser options for the Android platform will also not execute the script. As a result, while this type of attack may not work in purely Microsoft/Internet Explorer environments, it will be effective against Linux, Mac OS X, iPhones, iPads, some Android-based phones/tablets making, it an effective threat against most corporate environments. In fact, according to data compiled by statcounter.com, the combination of Firefox, Chrome, Opera and Safari make up a total of 54% of the web browser usage throughout the world, making this type of attack particularly concerning. (Usage Share of Web Browsers, 2011) Furthermore, while Windows computers running only Internet Explorer would be immune from this threat vector, Windows computers running Chrome, Firefox or Opera are still susceptable. As it is typically the more technical employees (e.g. IT personnel) who install alternate web browsers, when this attack vector is successful, it is likely to provide more value to the attacker.

3.4. Web Application Attacks

A variation of the browser attacks against end users involves using base64 to attempt to bypass web application security controls such as data input validation and web application firewall technology. While many such controls are configured to detect obvious JavaScript as part of their cross site scripting prevention capabilities, some may not detect a similar attack expressed in base64 such as <META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">. This type of attack is of particular concern as the target of cross-site scripting is often not the vulnerable application but the users of that application. Thus, while an organization's web applications may be completely secure, other applications used by their users may not be resulting in the potential for compromise.

3.5. Malware

Botnets are one of the more common forms of malware. They consist of many (often thousands or more) slaves or zombies that are centrally controlled by one or more master(s). Originally, IRC was used for control as it allowed many slaves to join a specific IRC channel to receive commands. As IRC is not often used in corporate environments, it was fairly easy to simply block outboud IRC access to mitigate the botnet risk. As a result, malware authors moved to HTTP for command and control. This is often done by placing HTML comments on a web page. These comments are not visible when casually browsing the page but can be seen when viewing the page's source code. The malware on the infected hosts is configured to periodically look for commands "hidden" as these HTML comments. The individual in control of the botnet simply updates the hidden comments to send new instructions to their zombies. (Team Cymru, 2008)

If these instructions were passed "in the clear", with no obfuscation, it would be easy for IDS/IPS systems to detect them. This would increase the likelihood of detection and make it much easier for malware analysts or incidnet responders to combat the problem. As a result, the instructions are often encoded using base64. The zombie has a built in base64 decoder that can be used to translate the instructions into commands that can be understood and executed by the zombie. While base64 is not the only encoding

used, it is common likely because it is fairly difficult to detect using automated means while not suffering from the processing overhead involved with true encryption.

4. Base64 Auditing

Given the risks associated with base64, having no program for detecting its use leaves an organization vulnerable to a variety of direct and indirect attacks. Given the complexities of detecting base64 however, such a program is an exercise in risk management and compromise. Detection systems must find a balance between excessive false positives and excessive false negatives but unlike some other types of detection, the elimination of both false positives and false negatives is not possible. In fact, any base64 detection solution is likely to include both. The goal is to reduce them to the extent possible.

In addition, the detection system must be tuned such that the most critical and/or accurate detection signatures "fire" first. Signatures that detect more than the presence of base64, such as the Emerging Threats rule for detecting basic web authentication should be configured to alert first, followed by more specific base64 detection.

4.1. Compromise

As discussed previously, planning base64 detection is an exercise in compromise. While regular expressions such as "[0-9a-zA-Z+/=_]{20,}" will detect virtually all base64 over 20 bytes in length, it will also result in significant false positives and should only be used in specific circumstances. More targetted regular expressions such as "(?:[A-Za-z0-9+/]{4}){2,}(?:[A-Za-z0-9+/]{2}[AEIMQUYcgkosw048]=|[A-Za-z0-9+/][AQgw]==)" will have fewer false positive results but will miss approximately one third of the base64 they see as they are looking for trailing equal signs. To address these concerns, an active program of base64 detection must be employed as follows:

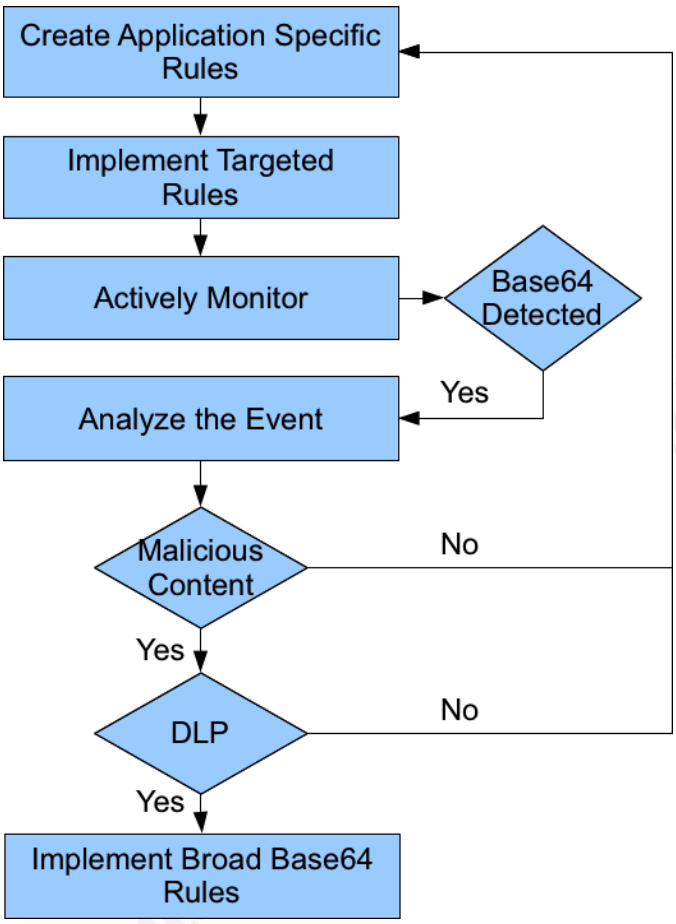
- Application specific base64 detection, such as the basic web authentication rule, should be used whenever possible.
- Targeted rules such as those using regular expressions that look for trailing equal signs should be used as high-level alerts.
- IDS operators should review alerts and add specificity to signatures as possible, thereby creating additional application sepcific base64 detection signatures.
- In the event that data exfiltration or targeted attacks are suspected, signatures using regular expressions that result in high false positive results should be employed but should be made as specific to source and destination IP address and port, traffic direction, etc. as possible.

Application specific, targeted base64 detection would include any signatures designed to look for protocols, such as basic web authentication, that utilize base64. While few of this type of signatures may be available to begin with, as base64 is detected crossing the network, the circumstances involving its use can be investigated and categorized as "known good" and "known bad". Signatures for "known good" base64 usage can be created to simply allow or ignore that traffic reducing the noise generated by the system. Similarly, specific signatures looking for unique characteristics (e.g. source address, destination address, source port, destination port or packet payload) of

known bad traffic can be created. Thus, over time, the base64 detection solution will become more accurate as it gets tuned to the specifics of its environment.

In addition to application specific rules, a base64 detection solution will need to have general rules for detecting base64 anywhere in any packet, regardless of protocol, such as those discussed previously in this document. Ideally, these detection signatures would be geared towards reducing false positive results. During this stage of the detection process, it is better to miss some base64 than to be overwhelmed with alerts. The rules provided previously in this document fit this pattern. They will detect base64 with either one or two trailing equal signs. This means that roughly two thirds of all base64 crossing the network will be detected. The goal at this phase is simply to broadly detect the use of base64 either entering or leaving the network. Any instances of base64 detected by these signatures should be investigated. The techniques for addressing known good and known bad base64 would then be used to create additional application specific rules.

If the use of base64 to circumvent DLP or to conduct specific attacks is detected, broad detection rules should then be implemented. These rules should leverage regular expressions such as [0-9a-zA-Z+/=_]{20,} that are highly subject to false positives but that would result in few, if any, false negatives. The regular expression should be used in an IDS rule that is specific in terms of traffic direction, source address, destination address, port an any other detail that can be used to reduce the volume of alerts. The goal at this phase is to catch everything related to the potential incident. The results should be investigated thoroughly and used, as appropriate, to pursue criminal, civil or administrative action and to update the application specific signatures. The following diagram provides a high level overview of this process:



This approach to detection is extremely active and requires knowledgeable responders and IDS administrators but is only appropriate for environments where some reasonable level of risk related to base64 is acceptable. Using this approach an attacker who is aware of the detection methods in place could plan their "attack" such that the input data would result in base64 without trailing equal signs. Also, when dealing with end user targeted attacks, missing one out of three base64 communications means that a significant compromise could occur without detection. This "accepted risk" approach to detecting base64 is, however, far better than simply ignoring the problem. In extremely high security environments, the use of broad detection rules could be used in place of the regular expressions that require the trailing equal signs. This approach would result in a high number of false positives but would only miss base64 smaller than 20 bytes.

Using these techniques, the detection of base64 can be customized to any organization and can be used to detect the majority of base64 threats regardless of source, application or protocol. Over time, these techniques can also result in a significant decrease in false positive and false negative results.

4.2. Snort Rules

In order to fully detect base64 using Snort, multiple rules are required, each designed for a specific purpose. A number of these rules are shown in the following table:

Use	Rule	False Alert Description
Used to detect base64 as part of basic web authentication.	alert tcp $HOME_NET any -> any $HTTP_PORTS (msg:"ET POLICY Outgoing Basic Auth Base64 HTTP Password detected unencrypted"; flow:established,to_server; content:"|0d 0a|Authorization|3a 20|Basic"; nocase; content:!"YW5vbnltb3VzOg=="; within:32; classtype:policy-violation; reference:url,doc.emergingthreats.net/bin/view/Main/2006380; reference:url,www.emergingthreats.net/cgi-bin/cvsweb.cgi/sigs/POLICY/POLICY_Basic_HTTP_Auth; sid:2006380; rev:10;) (Emerging Threats, 2011)	Low false positives but will miss all base64 not associated with basic web authentication
Used to detect "standard" base64 as well as base64 used for privacy-enhanced mail, MIME and Radix-64 encoding for OpenPGP and requires trailing equal sign	Alert tcp $HOME_NET and -> any any (msg:"Possible standard base64 detected"; pcre:"/ (?:[A-Za-z0-9+/]{4}){2,}(?:[A-Za-z0-9+/]{2}[AEIMQUYcgkosw048]=|[A-Za-z0-9+/][AQgw]==)/"; classtype:policy-violation; sid: XXXXXXXX;)	Minimal false positives but will miss all regular expressions without trailing equal sign.
Used to detect "standard" base64 as well as base64 used for privacy-enhanced mail, MIME and Radix-64 encoding for OpenPGP with no trailing equal sign required.	Alert tcp $HOME_NET and -> any any (msg:"Possible standard base64 detected"; pcre:"/ (?:[A-Za-z0-9+/]{4}){2,}(?:[A-Za-z0-9+/]{2}[AEIMQUYcgkosw048]|[A-Za-z0-9+/][AQgw])/"; classtype:policy-violation; sid: XXXXXXXX;)	High false positives.
Used to detect a modified version of base64 used for URL applications.	Alert tcp $HOME_NET and -> any any (msg:"Possible non-standard base64 detected"; pcre:"/ (?:[A-Za-z0-9\-_]{4}){2,}(?:[A-Za-z0-9+/]{2}[AEIMQUYcgkosw048]|[A-Za-z0-9\-_][AQgw])/"; classtype:policy-violation; sid: XXXXXXXX;)	High false positives.
Used to detect long ASCII strings with base64 compliant characters.	Alert tcp $HOME_NET and -> any any (msg:"Possible standard base64 detected"; pcre:"/[0-9a-zA-Z+/=_]{20,}/"; classtype:policy-violation; sid: XXXXXXXX;)	Extremely high false positive results.

5. Conclusion

Base64 represents a very real risk to organizations that rely on computers, networking and the Internet for a variety of reasons. Base64 is often used in place of encryption to transmit sensitive information including usernames and passwords which can result in unauthorized disclosure. Base64 can also be used to obfuscate attacks in an attempt to bypass detection and protection technologies. Unfortunately, the detection of base64 is extremely difficult as base64 is simply ASCII text that just happens to decode into something else. While the detection of base64 should be part of any monitoring program, it is always going to be an act of compromise involving reducing but not eliminating false positive and false negative results. To achieve the highest overall detection fidelity, organizations must implement an active program of detection that involves continual reviewing of alerts and tuning of the system. If done properly however, base64 detection can become an effective component of an overall information security program.

6. References

2006280. (2011, May 25). *Emerging Threats*. Retrieved August 16, 2011, from doc.emergingthreats.net/2006380

Coyier, C. (2010, March 25). DataÂ URIs | CSS-Tricks. *CSS-Tricks*. Retrieved August 16, 2011, from http://css-tricks.com/5970-data-uris

Craig. (2007, August 7). Filtering base64 encoded spam | Small Dropbear. *Small Drop Bear* . Retrieved August 16, 2011, from http://enc.com.au/2007/08/filtering-base64-encoded-spam

Franks. (n.d.). RFC2617 - HTTP Authentication. *Internet Engineering Task Force*. Retrieved August 16, 2011, from tools.ietf.org/html/rfc2117

Freed, N. (n.d.). Multipurpose Internet Mail Extensions. *Internet Engineering Task Force*. Retrieved August 16, 2011, from tools.ietf.org/html/rfc2045

Good, G. (n.d.). The LDAP Data Interchange Format (LDIF) - Technical Specifications. *Internet Engineering Task Force*. Retrieved August 16, 2011, from www.ietf.org/rfc/rfc2849.txt

Password Recovery Speeds. (2009, July 10). *Lockdown.co.uk - The Home Computer Security Center* . Retrieved August 16, 2011, from http://www.lockdown.co.uk/?pg=combi

Perera, H. (2011, May 1). History of Steganography. *Hareendra's Blog*. Retrieved August 16, 2011, from http://hareenlaks.blogspot.com/2011/04/history-of-steganography.html

Prabhakar, A. (2011, January 11). the Digital me: Base 64 Encoding. *the Digital me*. Retrieved August 16, 2011, from http://digitalpbk.blogspot.com/2006/12/base-64-encoding.html

Cymru. (n.d.). A Taste of HTTP Botnets. *Team Cymru*. Retrieved August 16, 2011, from www.team-cymru.com/ReadingRoom/Whitepapers/2008/http-botnets.pdf

Usage share of web browsers - Wikipedia, the free encyclopedia. (2011, July 26). *Wikipedia, the free encyclopedia*. Retrieved July 26, 2011, from http://en.wikipedia.org/wiki/Usage_share_of_web_browsers