FORCEPOINT
POWERED BY Raytheon

Protecting the human point.

DanderSpritz/PeddleCheap Traffic Analysis

FORCEPOINT RESEARCH REPORT

JOHN BERGBOM, SENIOR SECURITY ANALYST, FORCEPOINT

Table of Contents

Introduction

Although most firewalls can now recognize an initial DoublePulsar infection, spotting dormant implants where the initial infection took place before the April 2017 Shadow Brokers dump requires the ability to recognize implant communication.

Many exploits for recent vulnerabilities, payloads, and tools from the April 2017 Shadow Brokers dump have been extensively analyzed (see the References section for a list of resources). However, network traffic after an implant has been uploaded to a DoublePulsar backdoor has not been greatly analyzed to our knowledge. In this paper, we analyze how the PeddleCheap module of the DanderSpritz post-exploitation tool communicates with implants running on compromised machines.

To establish a context for this research, this is the high-level workflow:

1. Victim is infected with the DoublePulsar backdoor using the EternalBlue exploit
2. Malicious implant is created and configured
3. A PeddleCheap listener is started in the DanderSpritz GUI
4. Implant is uploaded to the victim via the DoublePulsar backdoor
5. Implant is executed and starts communicating with PeddleCheap

The focus of this research is on the last point: how PeddleCheap and the implant communicate.

Forcepoint™ Next Generation Firewall (NGFW), Forcepoint Web Security, and Forcepoint Web Security Cloud recognize and protect against traffic between PeddleCheap and malicious implants. Our purpose with this research is to help the security community combat the threat of dormant implants by providing fingerprints for use in intrusion detection/prevention systems. This research also provides insight into how a well-resourced intelligence agency may implement encrypted communication.

We also released a script [1] that can parse a traffic-capture of implant installation and subsequent implant communication with PeddleCheap as well as decrypt the communication.

Configuration and setup

DanderSpritz is a graphical user interface written in Java. It functions as a remote access tool for managing compromised computers. DanderSpritz has several modules, one of which is named PeddleCheap. Technically, it is the PeddleCheap module that communicates with implants that run on compromised hosts. For the rest of the document, we will refer mostly to PeddleCheap.

Details of how to use the EternalBlue exploit to infect the victim with DoublePulsar are left out, as there are plenty of resources describing the procedure. Likewise, information on how to upload an executable to DoublePulsar will be left out. For hands-on information describing both, see the article “How to Exploit EternalBlue & DoublePulsar to get an Empire/Meterpreter Session on Windows 7/2008” by Sheila A. Berta [2].

In brief, our setup configures PeddleCheap to listen to the HTTP port on the DanderSpritz host. The implant running on the victim initiates the communication by contacting this port. There are a few differences between HTTP and standard TCP traffic that will be pointed out below. More details on the PeddleCheap and implant configuration used in this research can be found in Appendix 1.

As an aid for other researchers, Appendix 3 contains some of the results of the PeddleCheap executable reverse-engineering done as part of this research.

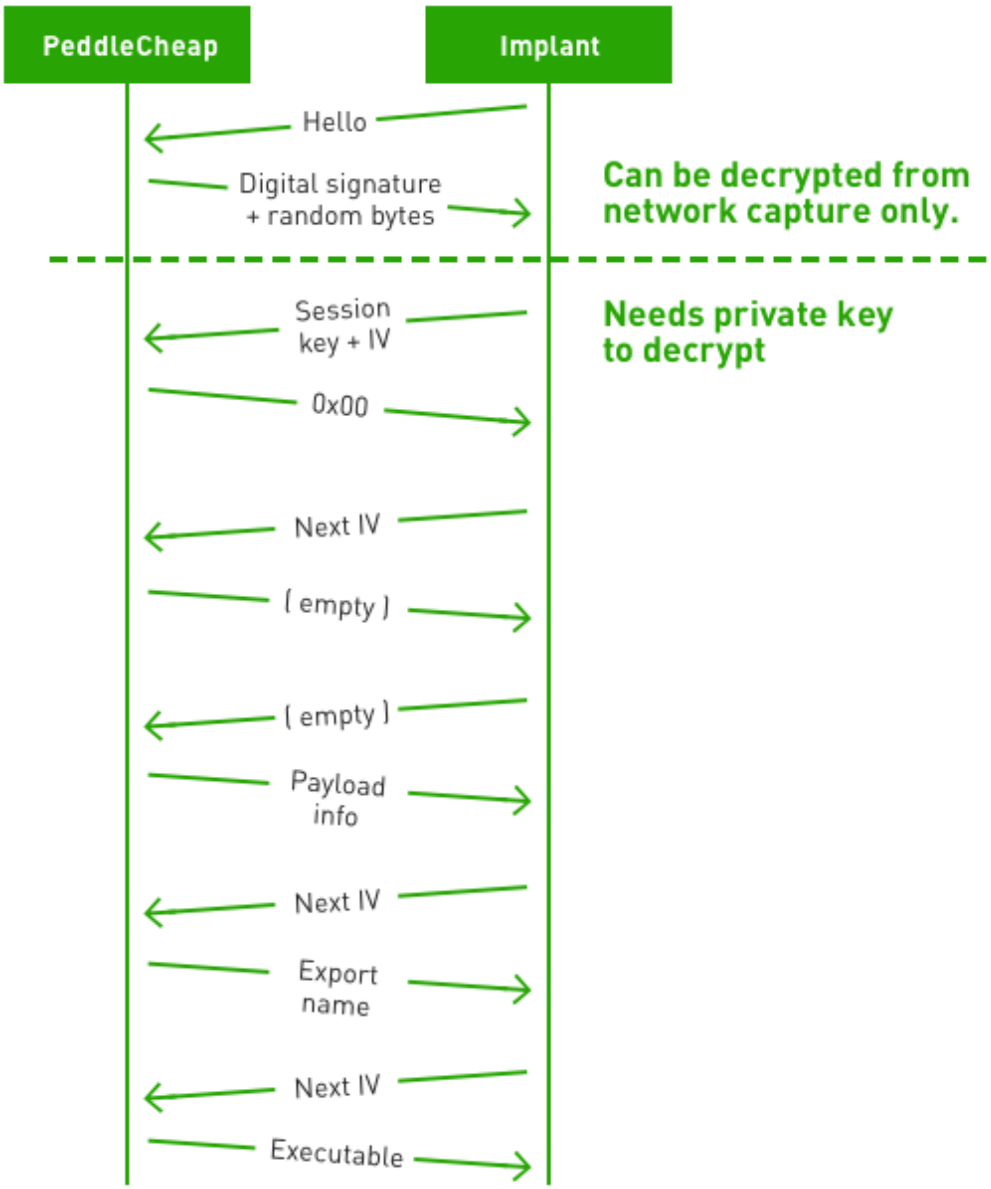
In the following section, we will dive into the PeddleCheap and implant communication.

PeddleCheap and implant communication

After DoublePulsar has received an executable to run, it will execute the implant which in turn will start communicating with PeddleCheap. This communication is totally unrelated to DoublePulsar.

According to our setup, the implant makes an HTTP connection at port 80 to a PeddleCheap listener running at the DanderSpritz host. While the HTTP headers are not encrypted, most of the HTTP payload is encrypted.

Communication starts with a three-way handshake. Here is a high-level sequence diagram of the traffic:



Decryption of traffic

Communication is split into two parts: a handshake and everything else. In the handshake, a digital signature proves to the implant that it is indeed communicating with PeddleCheap, and the implant provides a session key to PeddleCheap. The digital signature is encrypted using PeddleCheap's private key, and can be decrypted with the public key embedded in the implant (the private key is not embedded in the implant). This means the public key can be extracted from a network capture file and used to decrypt the digital signature. In the end of the handshake, the implant creates a symmetric session key and encrypts it using its public key. Possessing the private key is necessary to decrypt the session key. Since the private key stays in DanderSpritz and never goes across the wire, no traffic from the session key exchange and onwards can be decrypted from solely a network capture file.

For analysis purposes, the public and private keys can be obtained from the DanderSpritz host. They are named public_key.bin and private_key.bin, respectively. Their format is described in Appendix 2.

Symmetric session key

RSA key exchange is used to securely transfer a 128-bit session key from the implant to PeddleCheap. Once the session key has been exchanged, it is used for encrypting all subsequent communication. AES with Cipher Block Chaining (CBC) is used for the symmetric encryption.

Before each encrypted message is a clear-text header describing the size of the clear-text and a few other things. More details on this header can be found in the Message formats subsection.

Initialization vectors

The implant sends the first initialization vector (IV) to PeddleCheap in the same message that exchanges the symmetric session key. When sending messages back and forth, the encrypted payload P has two uses: it can be decrypted into a useful message, and it functions as an IV for decrypting the next payload (P+1). If one party sends two encrypted payloads in the same message, the first encrypted payload is used as an IV for decrypting the second. IVs are sent in clear-text and they are not reused.

Brief analysis of cryptographic implementation

As expected, the Shadow Brokers tools correctly implement cryptography as well as trust establishment. The following is a list of what the tools have implemented correctly with respect to PeddleCheap communicating with an implant:

- Establishing initial trust: implant sends a custom HTTP header that needs to match a certain configurable value (not used when using standard TCP)
- Digital signature: PeddleCheap encrypts a hardcoded string using the private key in DanderSpritz. This proves to the implant that it is communicating with PeddleCheap since only the owner of the private key would be able to make the encryption.
- Secure symmetric key exchange: implant creates a symmetric key and encrypts it with DanderSpritz's public key. This ensures that only PeddleCheap/DanderSpritz can decrypt it.
- IVs vary in an unpredictable way: encrypted message M functions as an IV for message M+1
- DanderSpritz's private key never goes across the wire (it is not embedded in the implant's code)
- Strong and theoretically sound encryption: AES encryption with CBC mode is used for encryption with a 128-bit IV and 128-bit session key

Message formats

The main analysis is done using the HTTP protocol. When standard TCP is used instead, the communication and encryption are essentially unchanged but there are a few differences in the message format. Below is a description of the message formats of requests and responses when HTTP is used. Differences in the formats for a standard TCP implant configuration are pointed out.

In addition to slight message format differences between HTTP and standard TCP, there is one slight difference in the message exchange: for standard TCP there is no initial “hello” message from the implant. See the example message exchange further below for more information.

See also Appendix 1 for more information on differences between HTTP and standard TCP.

Format of implant message to PeddleCheap when using an HTTP proxy

If using an HTTP proxy implant payload, implant messages to PeddleCheap start with an HTTP request:

```
POST / HTTP/1.1
Referrer: http://10.5.5.5/
TlEo: 0e59a2bc9:xxxxxxxx
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Win32)
Host: 10.5.5.5
Content-Length: <size>
Connection: Keep-Alive
Cache-Control: no-cache
```

Immediately after the last HTTP header come the bytes `\r\n\r\n`. All of the above is static, except for the Referrer, Host, and Content-Length header values as well as the hexadecimal sequence number marked in red above. The sequence number is in big-endian format starting with `0x00` and it is incremented by one for each message that the implant sends.

“`TlEo`” (marked in green above) is a custom HTTP header that must correspond to the connection prefix configured in PeddleCheap. A portion of the value of this header (marked in blue) is part of an implant ID (corresponding to `<Id>0xXXXXXXXXe59a2bc9</Id>` found in the DanderSpritz logs).

In some instances, for example the initial “hello” message, there is no more data in the message. However, the message typically continues:

`00000000`	`xx xx xx xx`	Size of clear-text message (in big-endian format)
`00000004`	`xx`	`0x00` if not symmetric key encryption. If symmetric key encryption, calculated value of: `0x10` − (size of clear-text message & `0x0f`). Total size of encrypted payload = purple DWORD + orange byte.
`00000005`	`xx`	`0x01` if symmetric key encryption is in use, `0x00` if not
`00000006`	`xx xx`	Unused
`00000008`	`<payload>`	Encrypted payload, reused as IV for encryption of the next payload

Sometimes there can be more than one payload tacked onto the same message. In that case, subsequent payloads use the same format, having eight bytes of meta-data followed by the payload.

Traffic always looks like this when the implant is configured to use an HTTP proxy. HTTP headers are always in clear-text no matter what port is used.

Format of implant message to PeddleCheap when using standard TCP

When the implant is configured to use standard TCP, the format of implant messages to PeddleCheap is similar to when using an HTTP proxy, except that the HTTP headers (and subsequent `\r\n\r\n` bytes) are missing. Since there are no HTTP headers, no sequence number is used.

The traffic always looks the same, no matter what port standard TCP is configured to use.

Format of PeddleCheap message to implant when using an HTTP proxy

PeddleCheap messages to the implant start like this:

```
HTTP/1.0 200 OK
Server: Microsoft-IIS/5.0
Content-Type: image/jpeg
Content-Length: <size>
```

Immediately after the last HTTP header come the bytes `\r\n\r\n`. Then comes:

`\x01\x00\x00\x00\xXX\xXX\xXX\xXX`

All of the above is static except for the Content-Length value and a sequence number marked in red. The sequence number is in little-endian format starting with `0x01` and it is incremented by one for each message that PeddleCheap sends.

In some instances there is no more data in the message, but typically the message continues:

`00000000`	`xx xx xx xx`	Size of clear-text message (in big-endian format)
`00000004`	`xx`	`0x00` if not symmetric key encryption. If symmetric key encryption, calculated value of: `0x10` − (size of clear-text message & `0x0f`). Total size of encrypted payload = purple DWORD + orange byte.
`00000005`	`xx`	`0x01` if symmetric key encryption is in use, `0x00` if not
`00000006`	`xx xx`	Unused
`00000008`	`<payload>`	Encrypted payload, reused as IV for encryption of the next payload

Sometimes there can be more than one payload tacked onto the same message. In that case, subsequent payloads use the same format, having eight bytes of meta-data followed by the payload.

Due to padding, the encrypted (and thus the decrypted) message may be longer than the original clear-text message. By specifying a size (marked in purple above), the implant will know how many of the decrypted bytes are actually in use.

Traffic always looks like this when the implant is configured to use an HTTP proxy. HTTP headers are always in clear-text, no matter what port is used.

Format of PeddleCheap message to implant when using standard TCP

When the implant is configured to use standard TCP, the format of PeddleCheap messages to the implant is similar to when using an HTTP proxy, but with the following differences:

- No HTTP headers (nor subsequent `\r\n\r\n` bytes)
- No static `\x01\x00\x00\x00` in the beginning
- No sequence number is used

The traffic always looks the same, no matter what port standard TCP is configured to use.

Used evasions

A host infected by DoublePulsar does not open any new ports, making DoublePulsar very stealthy. DoublePulsar uses the Server Message Block (SMB) protocol to function as a backdoor, but while doing so it continues to provide service for any benign use of the SMB port. This is an innovative technique.

When the PeddleCheap implant is copied onto the victim via DoublePulsar, the file transfer uses custom SMB extensions, including a reserved but unimplemented subcommand and the assignment of special meanings to several SMB fields. Valid responses to DoublePulsar commands are even concealed as error messages, leading a casual observer of the network traffic to believe the client sent some malformed data and the server responded with an error. Although some rudimentary encryption is added on top, this custom extension of the SMB protocol is an evasion technique that essentially hides the malicious traffic in plain sight.

When the PeddleCheap implant communicates with DanderSpritz using standard TCP, the implant resends data, appending more data in the resent packet. With these overlapping packets, part of the TCP stream is overwritten with the same data, and then more is appended. While in accordance with the TCP standard, this behavior is uncommon and could be an attempt to evade detection.

When the implant listens for incoming connections from DanderSpritz, it does not open a new port and listen on that port forever. Instead, it has a configured set of ports to listen on temporarily. The implant first listens on one port for some time, then closes the port and listens on another, rotating through all the ports in the set. Listening on ports in this manner is likely to draw less attention from monitoring systems than listening on a port indefinitely. Another benefit of using different ports is that it increases the chance of getting through any firewalls sitting between PeddleCheap and the operator.

Port knocking is also used to conceal the triggering of the PeddleCheap implant. Instead of opening any new listening ports, PeddleCheap monitors its host's network traffic and wakes up when a special sequence of "knocks" is detected.

When using HTTP, the response payload is binary data instead of clear-text HTTP. To ensure that the traffic is not blocked, this binary data is presented as an image (albeit not a validly formatted one). This is done by setting the Content-Type to `image/jpeg`.

Example message exchange using HTTP

Now that enough background has been provided, let us go through a series of message exchanges. We will look at the encrypted messages and see what is decrypted.

- **Note**: for brevity, the HTTP request and response headers will be excluded when showing the requests and responses

Request 1: “hello”

To initiate contact with PeddleCheap, the implant sends an HTTP POST message with an empty payload. The sequence number is 0. This “hello” message is the first part of a three-way handshake where a symmetric key is exchanged.

- **Note**: when using standard TCP there is no initial “hello” message. Instead, PeddleCheap sends the digital signature (see Response 1) immediately after the TCP handshake.

Response 1: digital signature / magic number

PeddleCheap response:

`00000000`	`01 00 00 00`	Static value (not sent when standard TCP is used)
`00000004`	`01 00 00 00`	Sequential number (not sent when standard TCP is used)
`00000008`	`00 00 01 00`	Size of clear-text message (always `0x0100` = 256)
`0000000c`	`00`	Size of encrypted payload = size of clear-text message
`0000000d`	`00`	No symmetric key encryption used
`0000000e`	`00 00`	Unused
`00000010`	`<payload>`	256 bytes of encrypted payload
`00000110`	`00 00 00 b9`	`0xb9` (185) random bytes will be sent (length can vary)
`00000114`	`00`	Zero since no symmetric key encryption is used
`00000115`	`00`	No symmetric key encryption used
`00000116`	`54 78`	Unused
`00000118`	`<random bytes>`	185 random bytes (in this case)

The payload is encrypted using DanderSpritz’s private key, meaning anybody can decrypt the message using the public key. To prove that the implant is actually communicating with PeddleCheap, a magic number inside the payload serves as a digital signature. Since only DanderSpritz possesses the private key, no other party would be able to encrypt a message that contains the magic number that the implant expects.

Using RSA to decrypt the 256-byte payload gives us the following (only the last 12 bytes of interest are detailed here):

`clear text = encrypted text ^ public key mod modulus =`		
`00000000`	`...`	
`000000f3`	`00 02 00 03`	DanderSpritz PeddleCheap version: 2.3.0
`000000f7`	`8e 30 71 ab`	Hardcoded magic number
`000000fb`	`dc 7d`	Two random bytes
`000000fd`	`00 b9`	Number of random bytes in next payload (`0xb9` = 185)

Note that `0xb9` in the clear-text is redundant information. The same information exists in the meta-data for the next payload.

When encrypting a specific clear-text message twice with RSA, the same crypto text is obtained. Due to the two random bytes at the end, there will be a variation in the crypto text between different runs.

Request 2: symmetric session key, implant platform data, and IV

Next, the target implant creates a symmetric key and sends it to PeddleCheap as an HTTP POST request:

`00000000`	`00 00 01 00`	Size of clear-text message (always `0x0100` = 256)
`00000004`	`00`	Size of encrypted payload = size of clear-text message
`00000005`	`00`	No symmetric key encryption used
`00000006`	`00 00`	Unused
`00000008`	`<payload>`	256 bytes of encrypted payload
`00000108`	`00 00 00 0c`	Size of clear-text message (`0x0c` = 12)
`0000010c`	`04`	Size of payload = `0x0c` + `0x04` = 16 bytes
`0000010d`	`01`	Symmetric key encryption used
`0000010e`	`00 00`	Unused
`00000110`	`<next IV>`	16-byte IV

The 16 bytes at the end give the next IV that PeddleCheap should use for AES encryption.

The 256-byte payload is encrypted using the public key of the implant. Since it can only be decrypted using the private key, traffic can no longer be decrypted from this point onward if we only have network captures available.

RSA decryption gives the following (only the last 48 bytes of interest are detailed here):

```
clear text = encrypted text ^ private key mod modulus =
00000000   ...
```

`000000d0`	`00 02 00 03`	Implant PeddleCheap version: 2.3.0
`000000d4`	`1f 00 00 00`	
`000000d8`	`9ac96bc2ca877c5394f1e5dcf009d07e`	Symmetric session key
`000000e8`	`0000000000000000`	Implant ID (specified when creating implant, 0 in this case)
`000000f0`	`00 00`	
`000000f2`	`00 08`	Implant architecture (8 = x64)
`000000f4`	`00 00`	
`000000f6`	`00 08`	Implant compiled architecture (8 = x64)
`000000f8`	`00 00`	
`000000fa`	`00 01`	Implant platform (1 = WinNT)
`000000fc`	`00 00`	
`000000fe`	`00 01`	Implant compiled platform (1 = WinNT)

Response 2: OS version check status

Now that a symmetric key has been exchanged, all communication from this point on uses AES encryption. PeddleCheap sends the following response:

`00000000`	`01 00 00 00`	Static value (not sent when standard TCP is used)
`00000004`	`02 00 00 00`	Sequential number (not sent when standard TCP is used)
`00000008`	`00 00 00 04`	Size of clear-text message
`0000000c`	`0c`	Size of encrypted payload = size of clear-text message + 12
`0000000d`	`01`	Symmetric key encryption used
`0000000e`	`c4 03`	Unused
`00000010`	`<payload>`	16 bytes of encrypted payload, reused as IV for next payload

AES decryption of the payload using the symmetric key and the IV given in the previous request gives `0x00000000`. PeddleCheap is hard coded to send `0x00000000` at this point in the communication. Based on the logs, the name of this response is “OS version check status,” hence the name of this subsection.

Request 3: OS version check status acknowledgment

The implant now sends new data to PeddleCheap:

`00000000`	`00 00 00 04`	Size of clear-text message
`00000004`	`0c`	Size of encrypted payload = `0x04` + `0x0c` = 16 bytes
`00000005`	`01`	Symmetric key encryption used
`00000006`	`00 00`	Unused
`00000008`	`<payload>`	16 bytes of encrypted payload, reused as IV for next payload

Decrypted with AES, the payload is `0x00000000`. The DanderSpritz logs inform us that this means the OS version check status has been received and stored.

Response 3: empty response

PeddleCheap sends a response containing only the static value and the incrementing sequence number:

`00000000`	`01 00 00 00`	Static value (not sent when standard TCP is used)
`00000004`	`03 00 00 00`	Sequential number (not sent when standard TCP is used)

- **Note 1**: when PeddleCheap uses standard TCP, this response is skipped. In that case we go directly to Response 4.
- **Note 2**: even when PeddleCheap uses an HTTP proxy, it sometimes skips this empty response and the subsequent query for the next command. In this case, it instead goes straight to sending the payload information to the implant (see Response 4).

Request 4: implant asks PeddleCheap for next command

The implant now sends an empty request asking PeddleCheap what to do next.

Response 4: PeddleCheap sends “PayloadInfo run type” and “file/library” information to implant

The DanderSpritz logs inform us that the response contains “PayloadInfo run type” information and “file/library” information:

`00000000`	`01 00 00 00`	Static value (not sent when standard TCP is used)
`00000004`	`04 00 00 00`	Sequential number (not sent when standard TCP is used)
`00000008`	`00 00 00 04`	Size of clear-text message
`0000000c`	`0c`	Size of encrypted payload = `0x04` + `0x0c` = 16 bytes
`0000000d`	`01`	Symmetric key encryption used
`0000000e`	`c8 04`	Unused
`00000010`	`<payload>`	16 bytes of encrypted payload (“PayloadInfo run type” information), reused as IV for next payload
`00000020`	`00 00 00 24`	Size of clear-text message (`0x24` = 36 bytes)
`00000024`	`0c`	Size of encrypted payload = `0x24` + `0x0c` = 48 bytes
`00000025`	`01`	Symmetric key encryption used
`00000026`	`01 00`	Unused
`00000028`	`<payload>`	48 bytes of encrypted payload (“file/library” information), reused as IV for next payload

Decrypted with AES, the “PayloadInfo run type” information is `0x00020000`.

Correspondingly, the clear-text “file/library” information is `0x00000000000000010000000000000001000000003dcce000000028af4205d000200080000`.

Request 5: implant acknowledges reception of “file/library” information

Implant acknowledges reception of “file/library” information by sending the following to PeddleCheap:

`00000000`	`00 00 00 04`	Size of clear-text message
`00000004`	`0c`	Size of encrypted payload = `0x04` + `0x0c` = 16 bytes
`00000005`	`01`	Symmetric key encryption used
`00000006`	`00 00`	Unused
`00000008`	`<payload>`	16 bytes of encrypted payload, reused as IV for next payload

Decrypted with AES, the acknowledgement is `0x00000000`.

Response 5: PeddleCheap sends “export name” to implant

According to the logs, the next piece of data PeddleCheap sends to the implant is an “export name”:

00000000	01 00 00 00	Static value (not sent when standard TCP is used)
00000004	05 00 00 00	Sequential number (not sent when standard TCP is used)
00000008	00 00 00 03	Size of clear-text message
0000000c	0d	Size of encrypted payload = 0x03 + 0x0d = 16 bytes
0000000d	01	Symmetric key encryption used
0000000e	c4 03	Unused
00000010	<payload>	16 bytes of encrypted payload (“export name”), reused as IV for next payload

Decrypted with AES, the “export name” is 0x233100.

Request 6: implant acknowledges reception of “export name”

Implant acknowledges reception of the “export name” by sending the following to PeddleCheap:

00000000	00 00 00 04	Size of clear-text message
00000004	0c	Size of encrypted payload = 0x04 + 0x0c = 16 bytes
00000005	01	Symmetric key encryption used
00000006	00 00	Unused
00000008	<payload>	16 bytes of encrypted payload, reused as IV for next payload

Decrypted with AES, the acknowledgment is 0x00000000.

Response 6: PeddleCheap sends file to execute to implant

Next, PeddleCheap sends an executable file in a compressed format to the implant:

00000000	01 00 00 00	Static value (not sent when standard TCP is used)
00000004	06 00 00 00	Sequential number (not sent when standard TCP is used)
00000008	00 02 8a f4	Size of clear-text message (0x028af4 = 166,644 bytes)
0000000c	0c	Size of encrypted payload = 0x028af4 + 0x0c = 166,656 bytes
0000000d	01	Symmetric key encryption used
0000000e	c4 03	Unused
00000010	<payload>	Encrypted first part of executable file

Decrypting with AES gives (snippet shown):

```
00000000  02 00 00 00 00 02 8a f4 00 05 2c 00 00 00 00 00  |..........,.....|
00000010  fc 4d 5a 90 00 03 00 b6 04 06 cb ff ff b8 b7 00  |.MZ.............|
00000020  01 40 92 00 af 08 01 7f 07 0e 1f ba 0e ff 00 b4  |.@..............|
00000030  09 cd 21 b8 01 4c 7f 09 54 68 69 73 ff 20 70 72  |..!..L..This. pr|
00000040  6f 67 72 61 6d ff 20 63 61 6e 6e 6f 74 20 ff 62  |ogram. cannot .b|
```

Interpretation:

- Red: size of compressed executable sent (same as seen above)
- Blue: original size of executable (`0x052c00` = 338,944 bytes in this case)
- Green: payload (the actual executable)

We see that the size of the sent executable (166,644 bytes) is a little less than half of the original executable (338,944 bytes), implying that the file is compressed using a packer of some kind. We also notice that the familiar “This program cannot be run...” in the executable has an `\xff` character inserted before each space character. This could bypass some pattern-matching security software when decrypted by the implant.

Fingerprinting

It is recommended to make separate signatures for HTTP and standard TCP traffic. Note that for either type, any port can be used.

Fingerprinting traffic with an HTTP proxy

While encryption is done well, nothing is perfect. Parts of the traffic are unencrypted, making fingerprinting relatively easy. The following traffic is sent in clear text:

- Hardcoded HTTP headers
- Static DWORD followed by a sequentially increasing number (DWORD) sent by PeddleCheap in the beginning of each HTTP payload
- Size of the clear-text message
- Byte telling whether symmetric key encryption is in use or not

In addition to the above, IVs are also sent in clear text, although this is not a problem from a cryptographic perspective nor from a fingerprinting perspective since IVs are not predictable.

Other useful properties for making fingerprints:

- Anomaly: there is a Content-Type mismatch in PeddleCheap’s messages to the implant: a Content-Type HTTP header of image/jpeg is used, although the message is not a valid JPEG file
- A custom HTTP header is used (by default `TlEo: 0e59a2bc9:000000xx`)

The PeddleCheap creators could easily have made the reply to the implant look like a valid JPEG file, removing the anomaly. Failure to do so was either a shortcoming or a conscious decision in order to save bandwidth.

One good way to recognize the traffic is to match for the digital signature from PeddleCheap:

00000000	01 00 00 00	Always this value
00000004	01 00 00 00	Always this value (in digital signature)
00000008	00 00 01 00	Always this value (in digital signature)
0000000c	00	Always this value (in digital signature)
0000000d	00	Always this value (in digital signature)
0000000e	xx xx	Could potentially be random
00000010	<payload>	Always 256 bytes, but content can vary
00000110	00 00	Always this value (in digital signature)
00000112	yy yy	Size of payload (which can vary)
00000114	00	Always this value (in digital signature)
00000115	00	Always this value (in digital signature)
00000116	zz zz	Could potentially be random
00000118	<random bytes>	“yy yy” (see above) number of random bytes

Fingerprinting traffic with standard TCP

It is more difficult to fingerprint traffic when standard TCP is used due to lack of HTTP headers, initial static DWORD, and sequence numbers. The best way to fingerprint this traffic is to look at the TCP payload and see if the payload starts with:

- DWORD in little-endian format (FIRST), followed by one byte (NEXT), followed by a byte having the value `0x00` or `0x01`, followed by two random bytes, followed by a payload (PAYLOAD) of size FIRST + NEXT
- Either the packet ends after PAYLOAD, or else the packet continues with another round of meta-data followed by a payload according to the above

Since TCP data is stream-based, fingerprinting done on the data stream will not be able to see the start and end of individual packets. This, as well as the fact that standard TCP has less static data and no sequence numbers, makes a case for fingerprinting standard TCP traffic differently than HTTP traffic.

We recommend fingerprinting standard TCP traffic by following the protocol structure far enough to avoid false positives. For example:

Implant sends symmetric key in the TCP data stream:

00000000	00 00 01 00	Always this value
00000004	00	Always this value
00000005	00	Always this value
00000006	xx xx	Could potentially be random
00000008	<payload>	Always 256 bytes, but content can vary
00000108	00 00 00 0c	Always this value
0000010c	04	Always this value
0000010d	01	Always this value
0000010e	xx xx	Could potentially be random
00000110	<next IV>	Always 16 bytes, but content can vary

The TCP stream in the same direction then continues with the implant sending an OS version check reception acknowledgment:

00000000	00 00 00 04	Always this value
00000004	0c	Always this value
00000005	01	Always this value
00000006	xx xx	Could potentially be random
00000008	<payload>	Always 16 bytes, but content can vary

Conclusions

PeddleCheap and its associated exploits were used by the intelligence community for years before they were leaked to the public. Simply recognizing and blocking initial PeddleCheap infections will not block systems that are already infected from communicating.

In our research, we devised a way to fingerprint PeddleCheap traffic without having to decrypt the traffic first. We can now recognize and block PeddleCheap based on certain characteristics of its encrypted traffic. This allows us to detect dormant implants on systems where the initial infection took place before the April 2017 Shadow Brokers dump.

Also, our research provides insight into how a well-resourced intelligence agency may implement malicious implants as well as command and control channels.

Future work

Some questions for future research include:

- Why does PeddleCheap send random bytes to the implant as part of the digital signature? Is it to ensure good encryption in the event that the implant is running in an environment where not enough entropy can be obtained?
- How does PeddleCheap compress executables before sending them to the implant, and how to decrypt them properly?
- How do PeddleCheap and the implant react if a firewall drops or rejects some message? Do they retry communication, or do they fail?
- Are there any practical cryptographic attacks against PeddleCheap and/or the implant? For example: CBC bit flipping attack, oracle padding attack, etc.
- How does PeddleCheap implement triggering by port knocking?

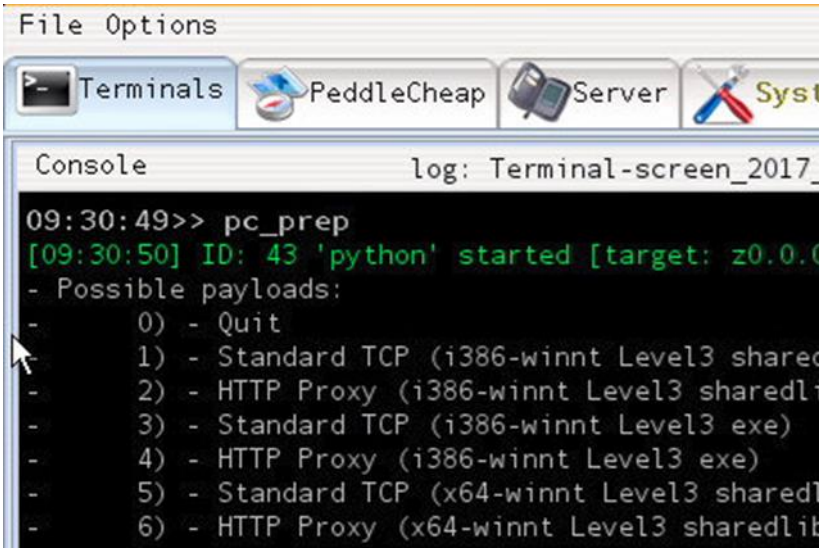
If you are considering exploring these or other related areas of research, we would be happy to hear from you. Get in touch with us at https://twitter.com/forcepointlabs.

Appendix 1: Configuration and setup

FuzzBunch/DanderSpritz was downloaded from https://github.com/fuzzbunch/fuzzbunch for the analysis.

Implant creation

The implant was created using the `pc_prep` command in DanderSpritz:



Option number 6 was chosen. Level 3 implants can be configured for either forward or reverse connections. (Level 4 implants can be configured to set up a trigger so that PeddleCheap can wake up the implant remotely.) Shared lib means it is a DLL instead of an EXE file. HTTP to port 80 was used, and the implant was set to make a reverse connection back to PeddleCheap. The implant was *not* configured with FelonyCrowbar.

In addition to configuration of Level 3 or Level 4, two other aspects of communication can be configured: to use HTTP requests and responses (HTTP proxy) or to use standard TCP. An HTTP proxy can be used for having the implant call back to PeddleCheap. It will always use clear-text HTTP headers, even if the HTTPS port is used.

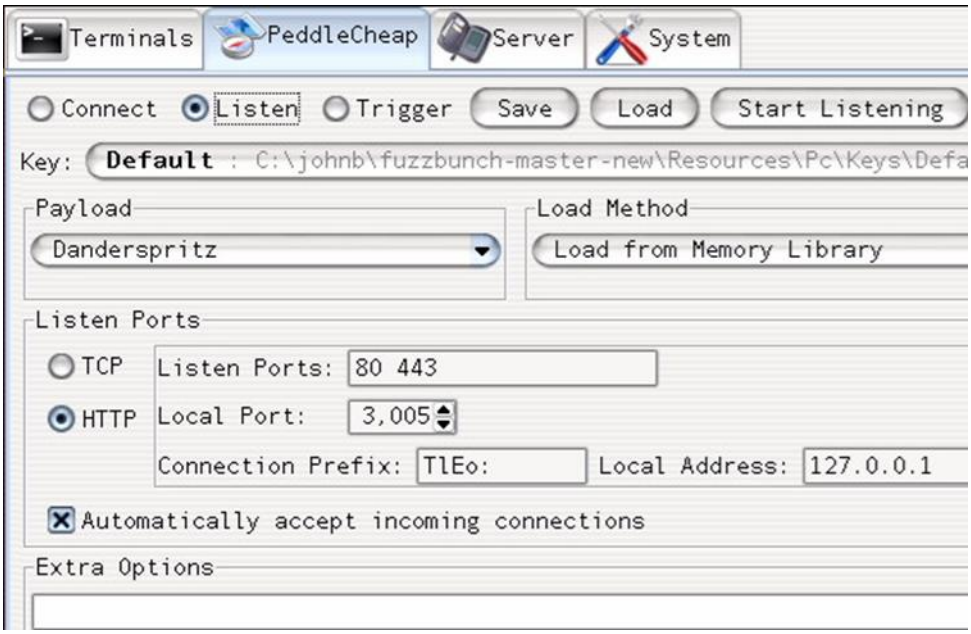
Standard TCP can be used for getting a reverse connection back to PeddleCheap as well as for having PeddleCheap make a forward connection to the implant. The only difference between using a forward or a reverse connection is which party makes the initial connection (and what default ports to use). Once the TCP handshake is done, the communication and encryption are identical.

For a forward connection, the implant has a configured set of ports to listen on. The ports are rotated so that the implant listens to one port for a while, then closes that port and starts listening to another port in the configured set.

Note that the generated implant contains an embedded public key. The corresponding private key will be kept in DanderSpritz only, and is never sent across the wire.

PeddleCheap configuration

The PeddleCheap configuration must conform to the configuration of the implant. In our case, we listen to HTTP port 80:



Note the `TlEo:` connection prefix. When using HTTP, the implant sends this connection prefix as an HTTP header when contacting PeddleCheap.

Appendix 2: Public and private key formats

The public and private keys can be obtained from the DanderSpritz host. They are named public_key.bin and private_key.bin respectively. These keys are only used for securely transferring a symmetric session key. Only the public key goes across the wire, whereas the private key exists only with DanderSpritz.

Public key file format

The public key is 516 bytes and it has the following format:

- Bytes 0-3: key length in number of bits
- Bytes 4-259: modulus in big-endian format
- Bytes 260-515: public key exponent in big-endian format. Its value is 2^16+1, written out as 256 bytes (`0x0000...010001`)

To extract the public key from a network capture, first we find the implant from the capture. We can then locate its embedded public key exponent since the exponent is known. From its starting offset, we can walk backwards 256 + 4 bytes to get the starting offset of the entire public key.

Private key file format

The private key is 1,412 bytes and it has the following format:

- Bytes 0-3: key length
- Bytes 4-259: modulus in big-endian format
- Bytes 260-515: public key exponent in big-endian format
- Bytes 516-771: private key exponent in big-endian format

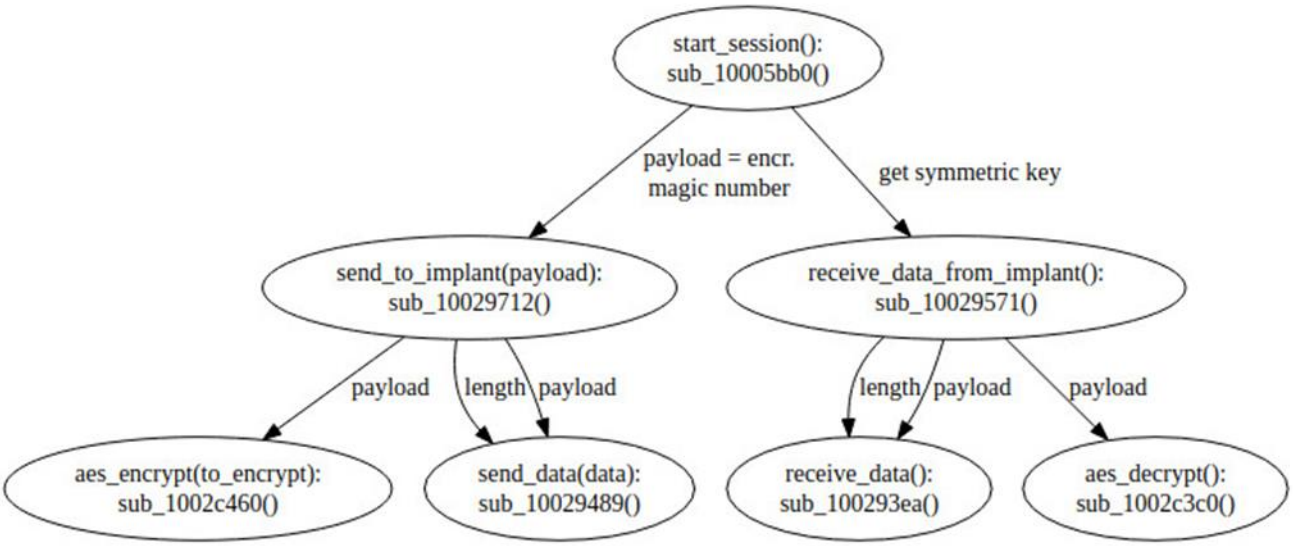
The rest of the private key contains the prime numbers, prime exponents, and the coefficient.

Appendix 3: Central code sections in PeddleCheap that handle communication

DanderSpritz is a Java-based program. It in turn runs the PE binary DszLpCore.exe, which listens for connections from implants. This PE binary imports a DLL named PeddleCheap_Lp.dll. The important code related to communication resides in this DLL. Let us do a brief overview of the most important code sections in this DLL for communicating with the implant.

Call graph

Below is a call graph for some central functions:



Functions

A selection of functions is listed below, including some of those shown in the call graph above:

- Main communication function: `sub_100061f0()`
- Starts session: `sub_10005bb0()`
- Sends data to implant: `sub_10029712()` → `sub_10029489()` → `sub_10029270()` → `sub_10029cb0()` → `sub_10029d5d()` → `sub_10027703()`
- Sends complete HTTP request to implant: `sub_10010550()` → `sub_10029270()` → `...`
- AES encryption: `sub_1002c460()`
- AES decryption: `sub_1002c3c0()`
- Receives data from implant: `sub_10029571()`
- Gets private key: `sub_10005710()`
- Encrypts magic number: `sub_1002b680()`
- Decrypts symmetric key: `sub_1002b670()`

More details about some of these functions follow in the subsections below.

Main communication function

On a high level, this function:

1. Starts a session
2. Gets remote OS information
3. Sends an executable to be run to the implant

Start session function

This function sets up the session between PeddleCheap and an implant. Briefly, it encrypts and sends a magic number to the implant, receives a symmetric key, and decrypts it with the private key. Below is abbreviated pseudo-code outlining this function:

```
func start_session() {
  // Get private key, encrypt the magic number and send to implant.
  magic_number = 0x8e3071ab;
  private_key = get_private_key();
  encr_magic_num = encrypt_magic_number(magic_number,private_key);
  send_data_to_implant(encr_magic_num);

  // Initiate random seed based on the current time, then fill a random-
  // sized buffer with random data and send to implant.
  seed = time64(0);
  srand(seed);
  nbr_rand_bytes = rand() % 256 + 128;
  i = 0;
  do
    rand_byte_buf[i++] = rand();
  while ( i < nbr_rand_bytes );
  send_data_to_implant(rand_byte_buf);

  // Receive a symmetric key from implant and decrypt it.
  encr_symm_key = receive_data_from_implant();
  symm_key = decrypt_symmetric_key(encr_symm_key, private_key);
}
```

Send data function

This function encrypts the data using AES encryption and then sends two pieces of information in the same message:

1. Size of decrypted payload
2. Actual encrypted payload

AES encryption function

This function encrypts the payload to send to the implant using AES encryption with CBC mode. The implant provides an IV to PeddleCheap, typically in the HTTP POST request. It takes the following parameters:

```
sub_1002c460(
      int session_key,
      int key_length,
      char *iv,
      int iv_length,
      char *clear_text,
      int clear_text_length,
      char *clear_text,
      int *clear_text_length_ptr)
```

References

[1]: https://github.com/johnbergbom/PeddleCheap/

[2]: Detailed How-To document describing how to run FuzzBunch for exploiting a target: https://dl.packetstormsecurity.net/papers/attack/exploiting-ebdp-en.pdf

[3]: DoublePulsar analysis: https://www.countercept.com/our-thinking/analyzing-the-doublepulsar-kernel-dll-injection-technique/

[4]: Overview of the tools in the leak: https://cysinfo.com/wp-content/uploads/2017/04/Shadow_release_updated.pdf

[5]: PeddleCheap analysis: https://www.countercept.com/our-thinking/analyzing-and-detecting-the-in-memory-peddlecheap-implant/

[6]: DanderSpritz overview: https://research.kudelskisecurity.com/2017/05/18/the-equation-groups-post-exploitation-tools-danderspritz-and-more-part-1/

[7]: DanderSpritz overview: https://danderspritz.com/

[8]: EternalBlue analysis: https://research.checkpoint.com/eternalblue-everything-know/

[9]: DanderSpritz overview: https://github.com/francisck/DanderSpritz_docs

[10]: EnglishmansDentist analysis: https://blogs.technet.microsoft.com/srd/2017/07/20/englishmansdentist-exploit-analysis/

[11]: EsteemAudit analysis: https://researchcenter.paloaltonetworks.com/2017/05/unit42-dissection-esteemaudit-windows-remote-desktop-exploit/

[12]: EsteemAudit analysis: https://blog.fortinet.com/2017/05/11/deep-analysis-of-esteemaudit

[13]: Analysis of patch for EsteemAudit: https://0patch.blogspot.fi/2017/06/a-quick-analysis-of-microsofts.html

[14]: Analysis of EternalRomance and EternalBlue and mitigations in Windows 10: https://blogs.technet.microsoft.com/mmpc/2017/06/16/analysis-of-the-shadow-brokers-release-and-mitigation-with-windows-10-virtualization-based-security/

[15]: EternalSynergy exploit analysis: https://blogs.technet.microsoft.com/srd/2017/07/13/eternal-synergy-exploit-analysis/

[16]: DoublePulsar analysis: https://www.countercept.com/our-thinking/doublepulsar-usermode-analysis-generic-reflective-dll-loader/

[17]: EternalChampion exploit analysis: https://blogs.technet.microsoft.com/srd/2017/06/29/eternal-champion-exploit-analysis/

[18]: EternalBlue analysis: http://blog.trendmicro.com/trendlabs-security-intelligence/ms17-010-eternalblue/

[19]: EternalBlue analysis: http://markus.co/memory-forensics/2017/06/04/eternalblue-smb.html

[20]: EternalBlue alternative implementation: https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/smb/ms17_010_eternalblue.rb

[21]: EternalBlue alternative implementation: https://github.com/worawit/MS17-010/blob/master/eternalblue_exploit7.py

[22]: EternalBlue alternative implementation: https://github.com/worawit/MS17-010/blob/master/eternalblue_exploit8.py

[23]: Porting EternalBlue to Windows 10: https://jennamagius.keybase.pub/EternalBlue_RiskSense-Exploit-Analysis-and-Port-to-Microsoft-Windows-10.pdf