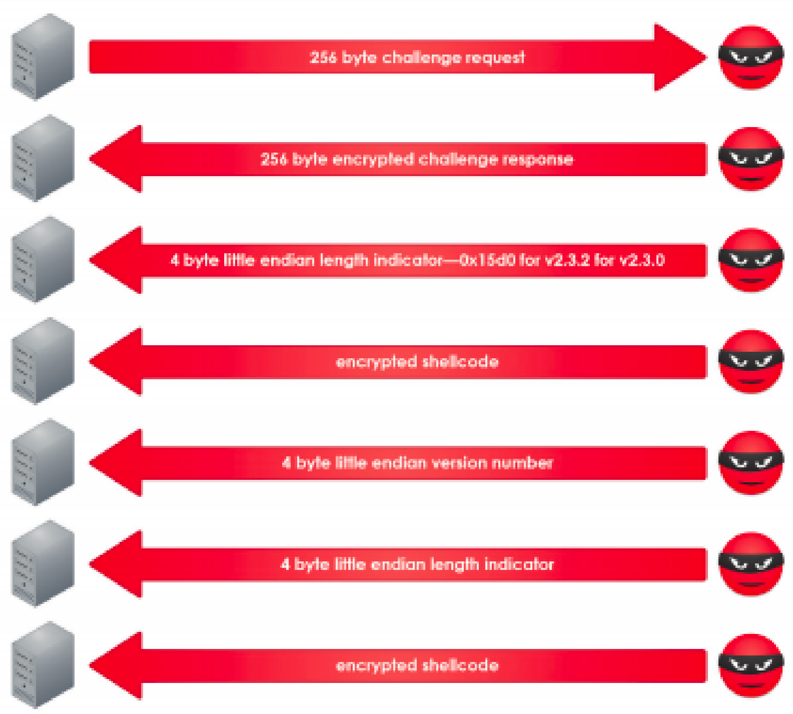
samvartaka

Crypto-trouble in Poison Ivy's C2 protocol

During the course of some research regarding the security of RAT C2 protocols and infrastructure i came across a stack buffer overflow disclosed in 2011 (OSVDB-83774) affecting the C2 server component of the Poison Ivy RAT. Poison Ivy (PIVY) is a 'golden oldie' RAT probably everyone in infosec circles is familiar with. Despite the fact that its last and final release (version 2.3.2) dates back to January 2008 it tends to resurface in various APT-style attacks now and then, likely owing to its ease of use, stability and full-featuredness. It probably also helps that off-the-shelf RATs used by everyone and their mother complicate attribution efforts a little bit. This RAT might be old and going out of style but it still pops up here and there (we all remember the RSA hack) so hey.

The buffer overflow in question was used by malware.lu to own the C2 infrastructure for the notorious APT1 which got them nominated for a Pwnie Award in 2013 in the category 'Epic 0wnage'. The details of this vulnerability and PIVY's C2 protocol have been discussed in great detail elsewhere (notably by Andrzej Dereszowski here and Gal Badishi here, here and here) but the takeaway is as follows. PIVY's C2 protocol is a connect-back protocol where the client (note that in RAT terminology client refers to the RAT instance running on the infected machine and server refers to the controlling end) connects to a predefined C2 server on a predefined port over TCP. See the following picture (courtesy of FireEye):

Figure 5: PIVY initial communication protocol


256 byte challenge request
256 byte encrypted challenge response
4 byte little endian length indicator—0x15d0 for v2.3.2 for v2.3.0
encrypted shellcode
4 byte little endian version number
4 byte little endian length indicator
encrypted shellcode

Note that it is the C2 server that authenticates itself to the infected client (and not the other way around). After the initial handshake is complete the C2 server is ready to receive client data (and the other way around) the first of which is an information message containing client details (username, operating system, service pack, etc.). Client data is encrypted and composed of a 0x20 bytes sized header followed by an arbitrary length body. It is in the parsing of this header that the (very suspicious, almost 'bugdoor'-like as Badishi notes) buffer overflow lies. The downside of this exploit as it has been covered up to now (as far as i can tell) is that it has to deal with the fact that PIVY traffic is encrypted. Since a lot of RATs use a 'static key' model where cryptographic keying material is often hardcoded into the client, recovery of a client binary corresponding to a particular campaign from an infected machine or via crowdsourcing efforts such as malwareconfig can overcome this. This doesn't go for all cases however and it is not unthinkable that client binaries prove unrecoverable or that a C2 server without corresponding clients is discovered. In such a scenario we are left without a cryptographic key which makes exploitation of this vulnerability problematic.

PIVY's Encryption Oracle

PIVY encrypts all its C2 traffic using the 256-bit variant of the Camellia block cipher in the ECB mode of operation. As mentioned above both client and server share a hardcoded key which is derived from an attacker-specified password (consisting of ASCII-printable characters only) as follows:

```
password + ("\x00" * (32 - len(password)))
```

Needless to say as far as password-based key derivation schemes are concerned this is a tad concerning if you are an attacker. This key is used both for challenge-response authentication of the server to the client and for bidirectional traffic encryption.

$$\begin{array}{l} \text{client} \rightarrow_c server \\ client \leftarrow_{E(c,k)} server \end{array}$$

The problem for someone seeking to exploit this buffer overflow on a C2 server for which they don't have the Camellia key is that they cannot reliably exploit it. While Badishi notes that all we need to get encrypted is the 0x20 bytes of client header (since this will trigger the buffer overflow after which control is redirected to our shellcode) and hence a brute-force 'spray and pray' approach (as incorporated in the official Metasploit module by means of the RANDHEADER variable) might (and sometimes does) work and while a captured handshake might give you a target for password bruteforcing this seems suboptimal in a scenario where one wants to limit interaction with a hostile C2 server as much as possible. The way the malware.lu folks exploited the PIVY C2 servers associated with APT1 was by probing them with a client challenge, recording the response (which corresponds to the Camellia ciphertext of the challenge) and brute-forcing it with a custom John the Ripper module to obtain the password so reliable exploitation could continue. But what if the APT1 operators had used a slightly stronger password than "pswpsw"?

Luckily the fact that the same static key is used for both challenge-response authentication and traffic encryption in PIVY effectively gives us an arbitrary encryption oracle. Considering that the challenge size is 256 bytes (yes, bytes, not bits) we have plenty of space to put whatever we want in there. Including our exploit payload thus allowing us to bypass the whole process of key recovery or unreliable exploitation attempts through randomized headers. Below is a fragment from my modified metasploit module which integrates the use of this encryption oracle:

```
# plaintext header
plaintextHeader = "\x01\x00\x00\x00\x01\x00\x00\x00\x00\x00\x01\x00\xbb\x00\x00\x00\xc2\x00\x00\x00\xc2\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"

# crafted challenge (first 32 bytes is our plaintext header), abuse challenge-response as encryption oracle
challenge = plaintextHeader + ("\x00" * (256 - 32))
sock.put(challenge)
# response = encrypt(challenge, key)
response = sock.get

# Cut and paste the first 32 bytes (our header inside the crafted challenge) without knowing the key
encryptedHeader = response[0, 32]
```

Do note that the fact that Camellia was used in ECB mode means that even if the challenge would be of a reasonable size (say equal to the Camellia blocksize of 128 bits) we would still have an arbitrary encryption oracle since we could simply stitch our ciphertext together from seperate challenge-response pairs. This fact allows us to turn the probablistic exploit implemented in the Metasploit module into a reliable, deterministic one which can be used against any PIVY C2 server regardless of whether we know the key or not (or how 'strong' the password it is derived from is).

Leaving the penguin showing through

Apart from the problems surrounding key management and key derivation in PIVY's C2 protocol its use of the ECB mode of operation leaves its 'penguin showing through' (as per Ben Nagy's Ode to ECB) which offers some other advantages to defenders (or attackers, the lines can blur here) as well.

$$\forall_{p_i,p_j}[p_i, p_j \in P, c \in C, p_i = p_j : (p_i \rightarrow c) \implies (p_j \rightarrow c)]$$

Under ECB mode identical plaintext blocks map to identical ciphertext blocks which gives us an extra measure of control in detecting the challenge-response handshake. Some approaches to detecting PIVY traffic (such as mentioned in this Trend Micro report) rely on detecting the exchange of two 256-byte high-entropy binary blobs). By checking whether identical blocks in the challenge map to identical blocks in the response potential false positives can be ruled out which might save time or increase monitoring performance. It also allows those scanning for PIVY C2 servers to rule out false positives by using specially crafted challenges (eg. with all blocks identical) and checking whether the ECB constraint holds on the response.

Attackers desperate to keep using PIVY could opt for migrating to older versions of the C2 server component in order to try avoiding getting exploited in the above fashion. That won't help much, however, as i've confirmed that the vulnerability also affects the following older versions of PIVY: 2.2.0, 2.3.0 and 2.3.1 and added them as targets (with the correct JMP ESP return addresses) to my updated Metasploit module. Versions before 2.2.0 are not affected by this particular vulnerability nor by the issues surrounding its usage of Camellia since PIVY prior to 2.2.0 uses RC4. Given that these date back to between 2005 and 2006 and have corresponding feature and platform support limitations that doesn't seem all that appealing either.

Either way i think the lesson here is that developers, even those who develop offensive security-related software, often have a hard time designing and implementing crypto protocols and in this case it could work to the advantage of an attacker wishing to *reliably* exploit a vulnerability.

My Metasploit module incorporating usage of the encryption oracle and adding additional targets can be found here.

← Previous Archive Next →

Published
07 September 2015

Category
malware

Tags

malware 1

exploitation 1

crypto 2

poisonivy 1