BLATSTING Command-and-Control protocol

laanwj.github.io/2016/09/04/blatsting-command-and-control.html

Laanwj's blog

Randomness

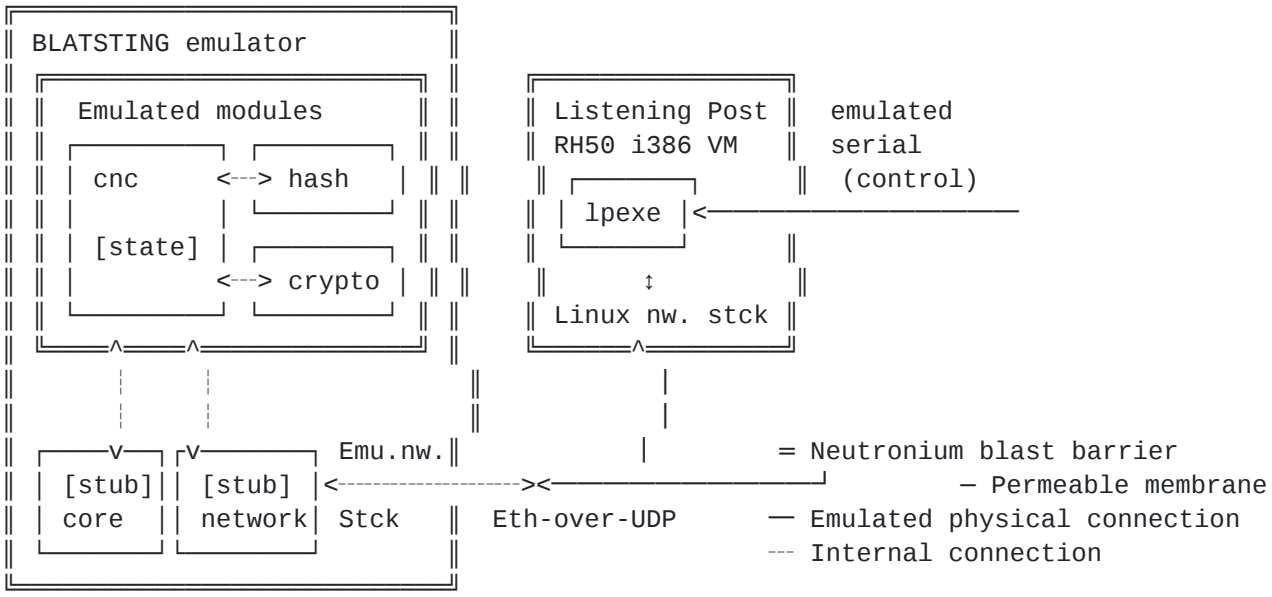
Blog About

In this installment I'm going to describe the Command-and-Control (or C&C) protocol of BLATSTING. This the protocol used in the network traffic between the malware and what is used by the person controlling it. I'm also going to see whether this traffic can be detected.

Setup

To figure out the details of how the communication works I've set up a simulation environment, which I've termed BEECHPONYZOO (named after BLATSTING's probable predecessor), where BLATSTING modules running in a simulation can communicate with the Listening Post executable `lp` running in a VM, in a way isolated from the real network and bare hardware:

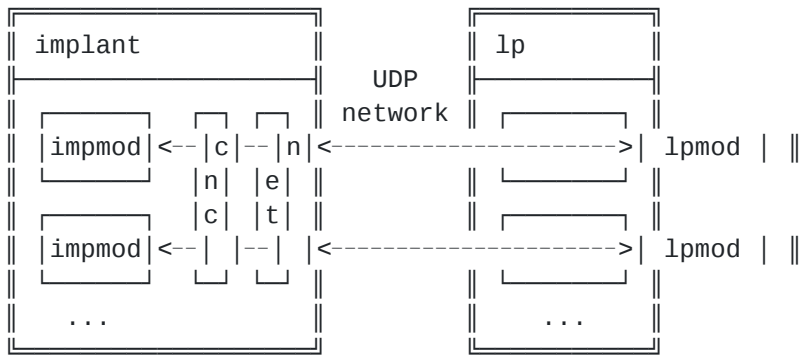
BEECHPONYZOO



See the gist for the entire module hierarchy, for the sake of expediency we're only emulating a part of it right now. But enough about my setup.

Network protocol

The implant (fancy spy word for rootkit) communicates with the Listening Post (spy for Command-and-Control program) using UDP datagrams over a IPv4 network. Those hoping to find IRC servers, bots and clients as in this presentation h/t: electrospaces.net) will unfortunately be disappointed.



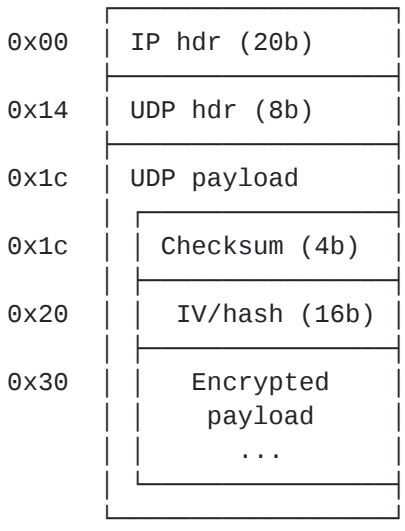
Interesting is that for the initial session setup, the source and destination IP address as well as ports do not matter. The only requirements for accepting the initial packets are that the size (including IP and UDP header, excluding Ethernet header) matches 68 or 72 bytes, and that a special checksum computed on the packet matches the one in the packet. This is a custom checksum, illustrated by the following C code:

```
/* cnc mystery packet checksum
 * arg1: 8 bytes, arg2: 4 bytes
 */
uint32_t cnc_checksum(uint8_t *arg1, uint8_t *arg2)
{
    const uint16_t i0 = arg1[0] | (arg1[1] << 8);
    const uint16_t i1 = arg1[2] | (arg1[3] << 8);
    const uint16_t i2 = arg1[4] | (arg1[5] << 8);
    const uint16_t i3 = arg1[6] | (arg1[7] << 8);
    const uint16_t j0 = arg2[0] | (arg2[1] << 8);
    const uint16_t j1 = arg2[2] | (arg2[3] << 8);

    const uint16_t b0 = ((i3 ^ j0) + (i0 ^ j1)) ^ arg1[2] ^ (arg1[7] << 8);
    const uint16_t b1 = ((i2 ^ j0) + (i1 ^ j1)) ^ arg1[5] ^ (arg1[4] << 8);
    const uint16_t b2 = ((i1 ^ j0) + (i2 ^ j1)) ^ arg1[0] ^ (arg1[1] << 8);
    const uint16_t b3 = ((i0 ^ j0) + (i3 ^ j1)) ^ arg1[3] ^ (arg1[6] << 8);
    return ((b3 ^ b1) << 16) ^ ((b3 * b0) << 5) ^ ((b1 * b2) << 11) ^ b2 ^ b0;
}
```

This checksum is checked for every C&C packet, not just for session setup. `arg1` points to the IV (more on this later), of which the first 8 bytes are checked. `arg2` is a 32-bit checksum key, stored internally and not part of the packet, assumed to differ per deployment. After checking, the inner payload is decrypted using RC6 in OFB mode. The key for RC6 is either a pre-shared key, or generated from a session challenge and the pre-shared key. This key is unrelated to the one used to key the checksum. The IV for OFB mode (16 bytes) is taken from the packet.

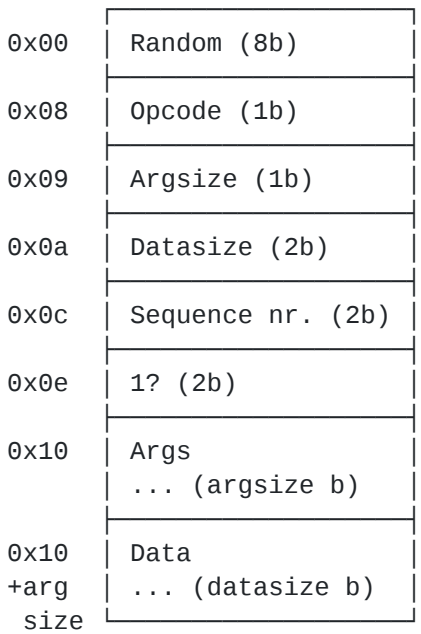
Packet format:



After decrypting the inner payload, it is hashed using SHA1 and the first 16 bytes of that hash are compared against the IV in the packet. If they match, the packet is accepted, otherwise it is rejected. This is “Authenticate and encrypt” and violates the Cryptographic

Doom Principle but I don't think the attacks described there are applicable. But who knows. Using plain SHA1 of the plaintext means that theoretically some information is leaked. Also the same crypto key is used in both directions.

The decrypted payload looks like this:



All values in this inner payload header are little-endian. Sometimes the actual args or data is big-endian, there is some inconsistency with endians at different nesting levels but this by far not the only protocol with this peculiarity.

Commands and replies use the same protocol, and share the same namespace of message codes. These are the various opcodes, helpfully taken from `lpexe` debug output:

Opcode	Argsize	Datasize	Name
0x00	4	4	HELLO
0x01	4 / 8	4 / 168	AUTH_RESP
0x02	0	16	CHALLENGE
0x03	?	?	ACK
0x04	?	?	ERROR
0x05	?	?	GOODBYE
0x06	?	?	BF_READ
0x07	?	?	BF_WRITE

Opcode	Argsize	Datasize	Name
0x08	?	?	MALLOC
0x09	?	?	FREE
0x0a	?	?	EXEC
0x0b	?	?	BOX_INFO
0x0c	?	?	BF_WRITE_STATUS
0x0d	?	?	MSG_FRAG
0x0e	?	?	MSG_ACK
0x0f	?	?	BF_FILE_STATE
0x10	?	?	BF_5i_CALL
0x11	?	?	BF_5i_REPLY
0x12	?	?	BF_5i_REQ_FRAG_PACKET

In the session negotiation phase, only `HELLO` (0) and `AUTH_RESP` (1) will be accepted. After a session is established the other calls can be used.

I haven't looked into detail of all the specific packets here. Some of it looks like the command set for a bare-bones post-exploitation shell, where the remote can allocate, read and write memory, inject code and execute it. Not all of the commands are implemented though, and the LP side doesn't even seem to use them. Maybe a remnant from an earlier protocol. Of note are mostly `BF_5i_CALL`, `BF_5i_REPLY`, these are used as a RPC system for communicating with loaded modules through their interfaces.

Detecting BLATSTING C&C packets

Now we get to the interesting part: can we detect BLATSTING C&C packets without knowing any secret key data? Remember that the session setup packets received by the implant are always UDP packets of 68 and 72 bytes (including IPv4 header), with arbitrary source and destination ports. The packets going the other way appear to have a fixed size too.

A session starts off like this:

```
LP          implant
 |------------>|       72 bytes (`HELLO`)
 |<------------|       80 bytes (`CHALLENGE`)
 |------------>|       72 bytes (`AUTH_RESP`)
 |<------------|      240 bytes (`AUTH_RESP`)
```

That's something that a IDS rule could be configured to look for. Another peculiarity of the outgoing packets (from the implant) is that they do not have the UDP checksum set (something that is allowed by RFC768). I don't know how common this is, but at least the Linux network stack computes it always.

Also remember the weird keyed checksum that we started with. It's not exactly a secure hash function and it's possible to check it without knowing the key. This could be done by brute-force (32-bit key space, but still that's a lot of overhead per packet) or by solving the equations, or by using a constraint solver such as Z3. I used the latter approach in z3_cnc_checksum.py. Given the checksum output and input it can check whether there is any possible key that will make it match the checksum.

Unfortunately this does have a ≈4% false-positive rate. By the way: it turns out that for this hash function the coverage of the output space for the same input with different keys is really small. E.g. for the same 8 byte arg1, trying all 2^32 arg2 possibilities will result in only 65495 different outputs. Due to (nearly) XORing them together the effect of both halves of `arg2` is very closely correlated: there is effectively only a 16 bit key space.

That's a bit disappointing, but by looking at the sizes of the packets as well as the checksum, the false-positive rate could be brought down. Or maybe by correlating multiple packets.

None of this helps to decrypt the packets, only detect them; as said before the key for RC6-OFB is distinct from the checksum key. The crypto key can only be extracted from the implant itself (safest way would be to dump kernel memory). Even though there are session keys, these do not offer any forward secrecy, so once this key is available all prior traffic can be decrypted.

Written on September 4, 2016

Tags: eqgrp malware
Filed under Reverse-engineering