5 An Advanced Mitigation Bypass for Packet-in-Packet; or, I'm burning 0day to use the phrase 'eighth of a nybble' in print.

by Travis Goodspeed
continuing work begun in collaboration with the Dartmouth Scooby Crew

Howdy y'all,

This short little article is a follow-up to my work on 802.15.4 packet-in-packet attacks, as published at Usenix WOOT 2011. In this article, I'll show how to craft PIP exploits that avoid the defense mechanisms introduced by the fine folks at Carleton University in Ontario.

As you may recall, the simple form of the packet-in-packet attack works by including the symbols that make up a Layer 1 packet at Layer 7. Normally, the interior bytes of a packet are escaped by the outer packet's header, but packet collisions sometimes destroy that header. However, collisions tend to be short and so leave the interior packet intact. On a busy band like 2.4GHz, this happens often enough that it can be used reliably to inject packets in a remote network.

At Wireless Days 2012, Biswas and company released a short paper entitled *A Lightweight Defence against the Packet in Packet Attack in ZigBee Networks*. Their trick is to use bit-stuffing of a sort to prevent control information from appearing within the payload. In particular, whenever they see four contiguous `00` symbols, they stuff an extra `FF` before the next symbol in order to ensure that the Zigbee packet's preamble and Start of Frame Delimiter (also called a Sync) are never found back-to-back inside of a transmitted packet.

So if the attacker injects `00 00 00 00 A7 ...` as in the original WOOT paper, Biswas' mitigation would send `00 00 00 00` *`FF`* `A7 ...` through the air, preventing a packet-in-packet injection. The receiving unit's networking stack would then transform this back to the original form, so software at higher layers could be none-the-wiser.

One simple bypass is to realize that the receiving radio may not in fact need four bytes of preamble. An upcoming tech report[2] from Dartmouth shows that the Telos B does not require more than one preamble byte, so `00 00 A7 ...` would successfully bypass Biswas' defense.

Another way to bypass this defense is to realize that 802.15.4 symbols are four bits wide, so you can abuse nybble alignment to sneak past Biswas' encoder. In this case, the attacker would send something like `F0 00 00 00 0A 7...`, allowing for eight nybbles, which are four misaligned bytes, of zeroes to be sent in a row without tripping the escaping mechanism. When the outer header is lost, the receiver will automatically re-align the interior packet.

But those are just bugs, easily identified and easily patched. Let's take a look at a full and proper bypass, one that's dignified and pretty damned difficult to anticipate. You see, byte boundaries in the symbol stream are just an accidental abstraction that doesn't really exist in the deepest physical layers, and they are not the only abstraction the hardware ignores. By finding and violating these abstractions—while retaining compatibility with the hardware receiver!—we can perform a packet-in-packet injection without getting caught by the filter.

You'll recall that I told you 802.15.4 symbols were nybble-sized. That's almost true, but strictly speaking, it's a comforting lie told to children. The truth is that there's a lower layer, where each nybble of the message is sent as 32 ones and zeroes, which are called 'chips' to distinguish them from higher-layer bits.

[2] Fingerprinting IEEE 802.15.4 Devices by Ira Ray Jenkins and the Dartmouth Scooby Crew, TR2014-746

The symbols and chip sequences are defined like this in the 802.15.4 standard. As each chip sequence has a respectably large Hamming distance from the others, an error-correcting symbol matcher on the receiving end can find the closest match to a symbol that arrives damaged.[3] This fix is absolutely transparent—by design—to all upper layers, starting with the symbol layer where SFD is matched to determine where a packet starts.

```
0 —— 11011001110000110101001000101110    8 —— 10001100100101100000011101111011
1 —— 11101101100111000011010100100010    9 —— 10111000110010010110000001110111
2 —— 00101110110110011100001101010010    A —— 01111011100011001001011000000111
3 —— 00100010111011011001110000110101    B —— 01110111101110001100100101100000
4 —— 01010010001011101101100111000011    C —— 00000111011110111000110010010110
5 —— 00110101001000101110110110011100    D —— 01100000011101111011100011001001
6 —— 11000011010100100010111011011001    E —— 10010110000001110111101110001100
7 —— 10011100001101010010001011101101    F —— 11001001011000000111011110111000
```

That is, the Preamble of an 802.15.4 packet can be written as either `00 00 00 00` or eight repetitions of the zero symbol `11011001110000110101001000101110`. While Biswas wants to escape any sequences of the interior symbols, he is actually just filtering at the byte level. Filtering at the symbol level would help, but even that could be bypassed by misaligned symbols.

"What the hell are misaligned symbols!?" you ask. Read on and I'll show you how to obfuscate a PIP attack by sending everything off by *an eighth of a nybble.*

I took the above listing, printed it to paper, and cut the rows apart. Sliding the rows around a bit shows that the symbols form two rings, in which rotating by an eighth of the length causes one symbol to line up with another. That is, if the timing is off by an eighth of a nybble, a 0 might be confused for a 1 or a 7. Two eighths shift of a nybble will produce a 2 or a 6, depending upon the direction.

```
0                          11011001110000110101001000101110 / 10001100100101100000011101111011 8
1                        1110110110011100001101010010001 / 10111000110010010110000001110111     9
2                      00101110110110011100001101010010 / 01111011100011001001011000000111       A
3                    00100010111011011001110000110101 / 01110111101110001100100101100000         B
4              010100100010111011011001110000011 / 00000111011110111000110010010110             C
5          00110101001000101110110110011100 / 01100000011101111011100011001001                   D
6      11000011010100100010111011011001 / 10010110000001110111101110001100                       E
7 10011100001101010010001011101101 / 11001001011000000111011110111000                           F
```

This technique would work for chipwise translations of any shift, but it just so happens that all translations occur in four-chip chunks because that's how the 802.15.4 symbol set was designed. Chip sequences this long are terribly difficult to work with in binary, and the alignment is convenient, so let's see them as hex. Just remember that each of these nybbles is really a chip-nybble, which is one-eighth of a symbol-nybble.

```
0      D9C3522E        8      8C96077B
1      ED9C3522        9      B8C96077
2      2ED9C352        A      7B8C9607
3      22ED9C35        B      77B8C960
4      522ED9C3        C      077B8C96
5      3522ED9C        D      6077B8C9
6      C3522ED9        E      96077B8C
7      9C3522ED        F      C96077B8
```

So now that we've got a denser notation, let's take a look at the packet header sequence that is blocked by Biswas, namely, the 4-bytes of zeroes. In this notation, the upper line represents 802.15.4 symbols, while the lower line shows the 802.15.4 chips, both in hex.

```
    0         0         0         0         0         0         0         0
D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E
```

As this sequence is forbidden (i.e., will be matched against by Biswas' bit stuffing trick) at the upper layers, we'd like to smuggle it through using misaligned symbols. In this case, we'll send 1 symbols instead

[3]Note that Hamming-distance might not be the best metric to match the symbol. Other methods, such as finding the longest stretch of perfectly-matched chips, will still work for the bypass presented in this article.

of 0 symbols, as shown on the lower half of the following diagram. Note how damned close they are to the upper half. At most one eighth of any symbol is wrong, and within a stretch of repeated symbols, every chip is correct.

```
     0         0         0         0         0         0         0         0
 D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E
    1         1         1         1         1         1         1         1
ED9C3522  ED9C3522  ED9C3522  ED9C3522  ED9C3522  ED9C3522  ED9C3522  ED9C3522
```

So instead of sending our injection string as `00000000A7`, we can move forward or backward one spot in the ring, sending `11111111B0` or `7777777796` as our packet header and applying the same shift to all the remaining symbols in the packet.

"But wait!" you might ask, "These symbols aren't correct! Between 0 and 4 chips of the shifted symbol fail to match the original."

The trick here is that the radio receiver must match *any* incoming chip sequence to *some* output symbol. To do this, it takes the most recent 32 chips it received and returns the symbol from the table that has the least Hamming distance from the received sample.

So when the radio is looking for A7 and sees B0, the error calculation looks a little like this.

```
BO —— 77B8C960D9C3522E
        ||||||||              <——Chips are nearly equal.
A7 ——  7B8C96079C3522ED
```

For the first symbol, the receiver expects the A symbol as `7B8C9607` but it gets `7B8C960`*D*. Note that these only differ by the last four chips, and that the Hamming distance between `0111` and `1101` is only two, so the difference between an `A` and a misaligned `B` in this case is only two.

It's easy to show that the worst off-by-one misalignment would make the Hamming distance differ by at most four. Comparing this with the distance between the existing symbols, you will see that they are all much further apart from one other. So we can obfuscate an entire inner packet, letting the receiver and a bit of radioland magic translate our packet from legal symbols into ones that ought to have been escaped.

Ain't that nifty?

— — — –   — — —   — – — –   — — —   – — –   — — –   —   – — — –   — — —

This technique of abusing sub-symbol misalignment to send a corrupted packet-in-packet which is reliably transformed back into a correct, meaningful packet should be portable to protocols other than 802.15.4.

For example, most Phase Shift Keyed (PSK) protocols can have phase misalignment that causes symbols to be confused for each other. Frequency Shift Keyed (FSK) protocols can have frequency misalignment when on neighboring channels, so that sometimes one channel in 2 FSK will see a packet intended for a neighboring channel, but with all or most of the bits flipped.

One last subject I should touch on is a fancy attempt by Michael Ossmann and Dominic Spill to defend against packet-in-packet attacks which was presented at Shmoocon 2014 and in a post to the Langsec mailing list. While they don't explicitly anticipate the bypass presented in this paper, it's worth noting that their example (5,2,2) Isolated Complementary Binary Linear Block Code (ICBLBC) does not seem to be vulnerable to my advanced bypass technique. Could it be that all such codes are accidentally invulnerable?

Evan Sultanik on the Digital Operatives Blog ported Mike and Dominic's technique for generating codes to Microsoft's Z3 theorem prover and came up with a number of new ICBLBC codes.

With so many to choose from, surely a clever reader could extend Evan's Z3 code to search just for those ICBLBC codes which are vulnerable to type confusion with misalignment? I'll buy a beer for the first neighbor to demo such a PoC, and another beer for the first neighbor to convincingly extend Mike and Dominic's defense to cover misaligned symbols. For inspiration, read about how Barisani and Bianco[4] were able to do packet-in-packet injections by ignoring Layer 1 and injecting at Layer 2.

Cheers from Samland,
—Travis

[4] Fully Arbitrary 802.3 Packet Injection: Maximizing the Ethernet Attack Surface by Andrea Barisani and Daniele Bianco at Black Hat 2013