Friday, January 8, 2016

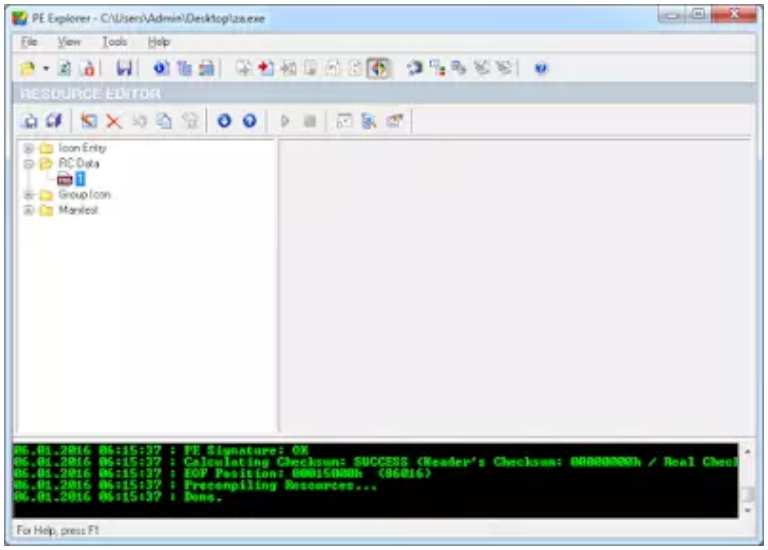
ZeroAccess 3 Analysis

After the takedown attempt in December 2013, the current status of ZeroAccess (alias Sirefef) has been much disputed. Initially the botmaster had released a command to the remaining infrastructure which contained nothing but the message "White Flag", signaling they may had given up on the botnet; but between March 21st 2014 and July 2nd 2014, SecureWorks had reported the ZeroAccess infrastructure resumed distributing commands.

Ever since the takedown ZeroAccess has not attempted to infect new hosts, instead it simply uses the large number of remaining infections to perform tasks—until now, that is. On January 3rd R136a1 came across a previously unseen sample, which FireF0X confirmed to be a variant of ZeroAccess. It's uncertain how long this sample has been in the wild, but it certainly points towards the botnet being a little less dead than was previously claimed.

Dropper

The first thing I noticed was the dropper has a resource file which contains a 61 KB PNG.



The image may look corrupted, which is because the developer decided to store the encrypted PE file in a valid PNG format. Rather than using stenography to hide the code in an existing image, the image *is* the code.

The dropper uses the native Ldr API to find and acquire the resource (`LdrFindResource_U`, `LdrAccessResource`) followed by the GdiPlus API to convert the PNG to a bitmap (`GdipCreateBitmapFromStream`), get the size of the code (`GdipGetImageWidth` × `GdipGetImageHeight` × 4), then decrypts it by xor'ing it with the key `'wDT:'`.

Execution is then passed to some shellcode which is pointed to by the `e_res` field of the decrypted PE's header. The shellcode resides directly after the `.reloc` section but is not part of the PE and therefore will not be present if the executable is later dumped after it's loaded.

```
mov     eax, [esp+58h+decrypted_pe]
mov     [esp+58h+ProcedureAddress], ebx
push    eax
mov     ecx, [eax+1Ch]  ; IMAGE_DOS_HEADER->e_res
add     ecx, eax
call    ecx             ; Shellcode entry point
mov     edi, eax
test    edi, edi
jz      short loc_4011BA
```

A unique and interesting thing about this shellcode is the way in which it handles resolving strings. Due to the fact shellcode is designed to be executed at whatever address it's allocated, it can't have the address of the strings hardcoded in; instead, it has to find their absolute address somehow. In this shellcode the author places a relative call directly before the string which makes a relative call to a stub after it, the stub then pops the return address into the eax and returns to the shellcode. As a result when `string_ZwMapViewOfSection` is called, the return address of call `loc_22E` (the first byte of the string 'ZwMapViewOfSection') ends up in the eax register.

```
string_ZwMapViewOfSection:

                call    loc_22E
; ---------------------------------------
aZwmapviewofsec db 'ZwMapViewOfSection',0
; ---------------------------------------

loc_22E:
                pop     eax
                retn
```

The shellcode itself seems pretty mundane: it simply sets a hardware breakpoint on `ZwMapViewOfSection` using `ZwSetContextThread` with flag set to `CONTEXT_DEBUG_REGISTERS`, then loads shellstyle.dll (a legitimate system DLL). What actually happens though is when the PE loader tries to map shellstyle into memory, the breakpoint is hit and the shellcode's exception handler is called. The exception handler then modifies the return address of `ZwMapViewOfSection` to point to a function which replaces the module with the one we saw decrypted earlier, as a result loading the malicious DLL in the place of shellstyle.dll.

Once the shellcode has replaced shellstyle.dll, execution is returned to the dropper which then uses `LdrGetProcedureAddress` to resolve and call the payload's entrypoint.

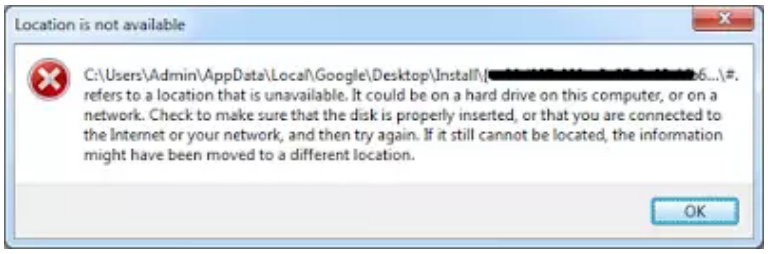
First Stage Payload

The payload consists of a DLL file with an embedded Microsoft Cabinet file (.cab), which can be found by searching the binary for the string 'MSCF'. The cabinet contains 6 files: 3 for 32-bit and 3 for 64-bit.

File	Description
s32	Boostrap list containing IPs and ports for 32-bit nodes
s64	Boostrap list containing IPs and ports for 64-bit nodes
k32	32-bit final stage payload
k32	64-bit final stage payload
l32	Shellcode used to load k32
l32	Shellcode used to load k64

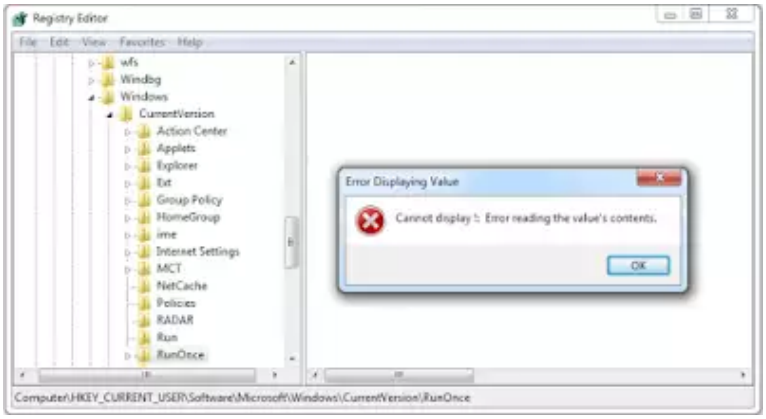
Next the payload checks a couple of events to see if the system is already infected, if not it create the event `\BaseNamedObjects\Restricted\{12E9D947-EDF5-4191-AADB-F51815F004D8}`. A semi-unique identifier for the bot is created by MD5 hashing the volume creation time of `\SystemRoot`, then formatting it as a 32-bit GUID (just like with older versions of the bot).

This version of ZeroAccess stores itself in a folder named `#.`, which is contained within the directory `%localappdata%\Google\Desktop\Install\<BotGUID>`. The folder name `#.` is special because it makes use of a difference in operation between the NativeAPI (ntdll) and Windows API (kernel32). Using native API functions a folder with a dot at the end of the name is entirely valid and
functional; however, under the Windows API it is not; the result is a usable folder that the malware can freely use to store and execute its components, but that cannot be accessed or modified by explorer or any application not using the Native API. For further protection, the folder is set as an NTFS Reparse Point which redirects to nowhere, preventing access by any means that is susceptible to NTFS re-parsing.

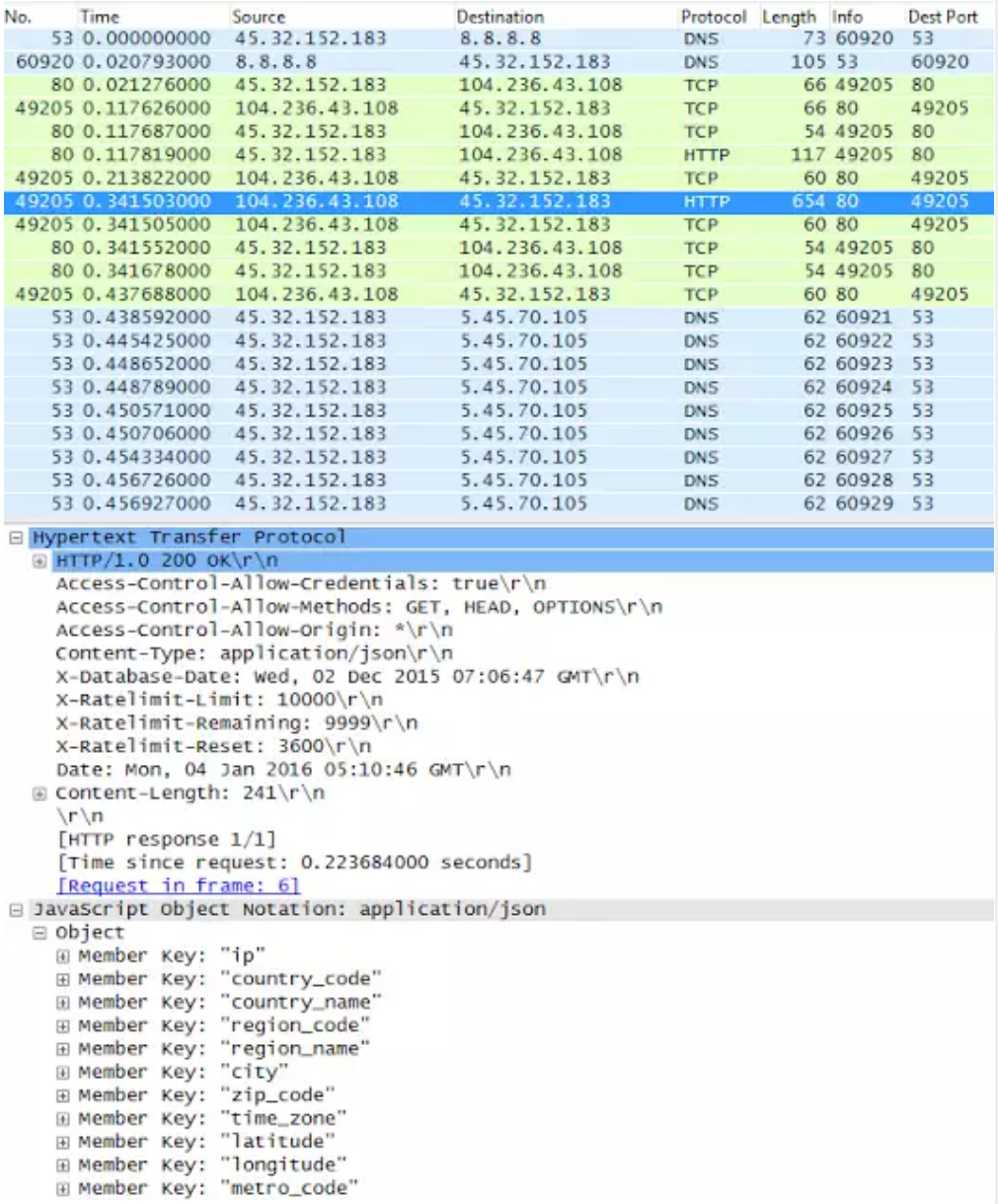


The payload extracts the bootstrap list (s32 or s64 depending on architecture) from the cabinet file, then decrypts and stores it inside the install directory in a file named `@`. The dropper is also moved to the same directory with the name `GoogleUpdate.exe`.

`GoogleUpdate.exe` is added to autostart by creating a key under `HKCU\Software\Microsoft\Windows\CurrentVersion\RunOnce` with the the bot's GUID as its name; the key name is prefixed with a null byte and written as Unicode, preventing Regedit from displaying it.



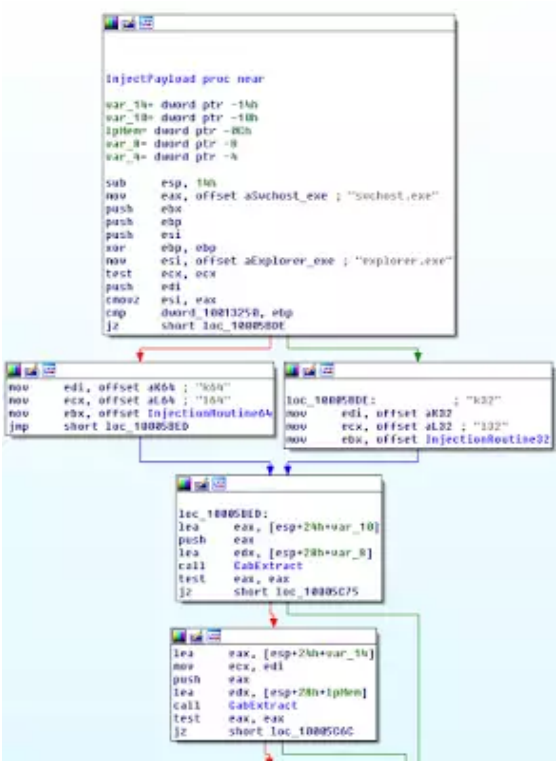
Throughout its installation process the bot sends status updates to a C&C via UDP on port 53 (DNS) with geolocation information obtained from the freegeoip.net api and an affiliate id stored in the bot (ZeroAccess uses an affiliate scheme in which people can sign up to be paid a small amount of money for each computer they infect with the malware). All status update communication is encrypted with the key `0x4C4F4E47` ('`LONG`'), which is the same key as in previous versions.



No.	Time	Source	Destination	Protocol	Length	Info	Dest Port
53	0.000000000	45.32.152.183	8.8.8.8	DNS	73	60920	53
60920	0.020793000	8.8.8.8	45.32.152.183	DNS	105	53	60920
80	0.021276000	45.32.152.183	104.236.43.108	TCP	66	49205	80
49205	0.117626000	104.236.43.108	45.32.152.183	TCP	66	80	49205
80	0.117687000	45.32.152.183	104.236.43.108	TCP	54	49205	80
80	0.117819000	45.32.152.183	104.236.43.108	HTTP	117	49205	80
49205	0.213822000	104.236.43.108	45.32.152.183	TCP	60	80	49205
49205	0.341503000	104.236.43.108	45.32.152.183	HTTP	654	80	49205
49205	0.341505000	104.236.43.108	45.32.152.183	TCP	60	80	49205
80	0.341552000	45.32.152.183	104.236.43.108	TCP	54	49205	80
80	0.341678000	45.32.152.183	104.236.43.108	TCP	54	49205	80
49205	0.437688000	104.236.43.108	45.32.152.183	TCP	60	80	49205
53	0.438592000	45.32.152.183	5.45.70.105	DNS	62	60921	53
53	0.445425000	45.32.152.183	5.45.70.105	DNS	62	60922	53
53	0.448652000	45.32.152.183	5.45.70.105	DNS	62	60923	53
53	0.448789000	45.32.152.183	5.45.70.105	DNS	62	60924	53
53	0.450571000	45.32.152.183	5.45.70.105	DNS	62	60925	53
53	0.450706000	45.32.152.183	5.45.70.105	DNS	62	60926	53
53	0.454334000	45.32.152.183	5.45.70.105	DNS	62	60927	53
53	0.456726000	45.32.152.183	5.45.70.105	DNS	62	60928	53
53	0.456927000	45.32.152.183	5.45.70.105	DNS	62	60929	53

Finally the main payloads `l32` and `k32` or `l64` and `k64` are extracted from the cabinet (my hypothesis is 'l' stands for 'Loader' and k stands for 'Kernel'). The ZeroAccess kernel is a DLL containing the main code for communicating with the peer-to-peer network and the loader is used to load the DLL in the same way the dropper loaded `shellstyle.dll`, except this time the DLL loaded and replaced is `comres.dll`.

If the process is UAC elevated it injects into `svchost.exe`, otherwise it will attempt to inject into explorer and bypass UAC using the same method as previous version (detailed in FireF0X's UACMe), before finally injecting into `svchost.exe`.



The injection methods works by writing the kernel, followed by the loader, to the remote process using `ZwAllocateVirtualMemory` and `ZwWriteVirtualMemory`, then executing the loader by calling `ZwQueueApcThread`. In the case of 64-bit, the payload will use Heaven's Gate to switch into 64-bit mode before calling the injection routine, allowing it to inject 64-bit processes from WOW64.

Peer to Peer Communications

The new ZeroAccess variant largely preserves the communication protocol of its previous versions, with some notable changes. For completeness, the UDP packet format for C&C communication has a common header of the form

```
struct zeroaccess_packet {
  uint32_t crc32;
  uint32_t type;
  uint32_t id;
  uint32_t length        : 16;
  uint32_t reply         :  1;
  uint32_t packet_number : 15;
};
```

The `packet_number` field appears to be new. It is used to ensure incoming packets match earlier requests, possibly to prevent '`retL`' flooding by takedown attempts. The `type` field can be one of '`getL`' and '`retL`' —the '`newL`' field of some earlier versions is now gone. `id` is a 32-bit bot session id, generated at startup and used to identify sessions. The reply field indicates whether the receiving node should reply with a `getL` of its own.

Every packet is encrypted—as before—with a very simple rolling-xor algorithm:

```
void zeroaccess_encrypt(unsigned char * message, size_t bytes) {
  uint32_t k = 0x31323334UL;
  assert(bytes % 4 == 0);
  for(size_t i = 0; i < bytes; i += 4) {
    store_le32(m + i, load_le32(m + i) ^ k);
    k = (k << 1) | (k >> 31);
  }
}
```

Note the new key, '`1234`', instead of the old one '`ftp2`'. This same algorithm is used with the '`LONG`' key mentioned above.

The `getL` packet is empty and consists only of the above header. The `retL` packet format is slightly changed due to one of the most noteworthy changes in the protocol—*UDP ports are now randomly-generated*, and thus the `retL` packets must include port information. The `retL` packet is of the form

```
struct peer_entry {
  uint32_t ip;
  uint32_t port : 14;
  uint32_t timestamp : 18;
};

struct file_entry {
  uint32_t id;
  uint32_t timestamp;
  uint32_t length;
  uint8_t  signature[128];
};

struct retL {
  peer_entry peers[16];
  file_entry files[/*variable*/];
};
```

On 32-bit Windows hosts, the port number is computed as `peers[i].port + 16384`, whereas on 64-bit Windows, it is computed as `peers[i].port + 32768`. It is therefore simple to figure out an individual node's platform by its UDP port number. The timestamp is computed as `(GetTime() - 0x41000000) / 3600`, where GetTime() is

```
ULONG GetTime() {
  FILETIME time;
  ULONG stamp;
  GetSystemTimeAsFileTime(&time);
  RtlTimeToSecondsSince1980((LARGE_INTEGER *)&time, &stamp);
```

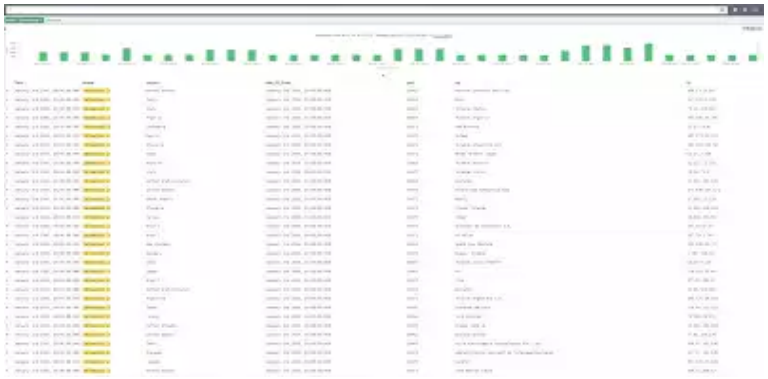
```
    return stamp;
}
```

We can see that the base timestamp `0x41000000`, i.e., July 22 2014, lower-bounds the age of this variant. The file list format remains unchanged (modulo timestamp format), albeit with a distinct RSA-1024 verification key compared to the previous botnets.

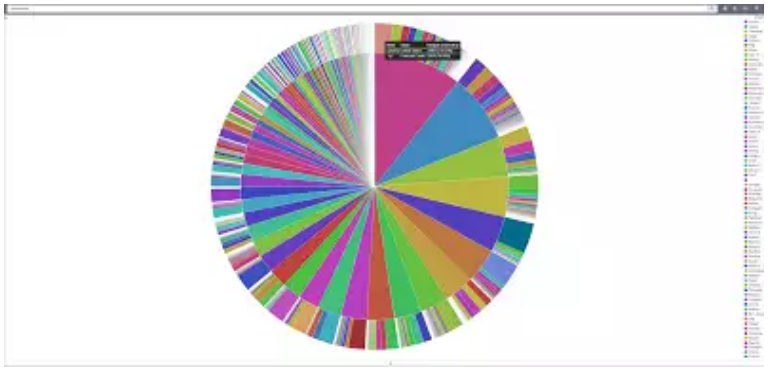
The bot must also remember what its port number is. This information is stored in the Extended Attributes of the `@` file. The list of known peers is stored in this file, as remarked above.

Statistics

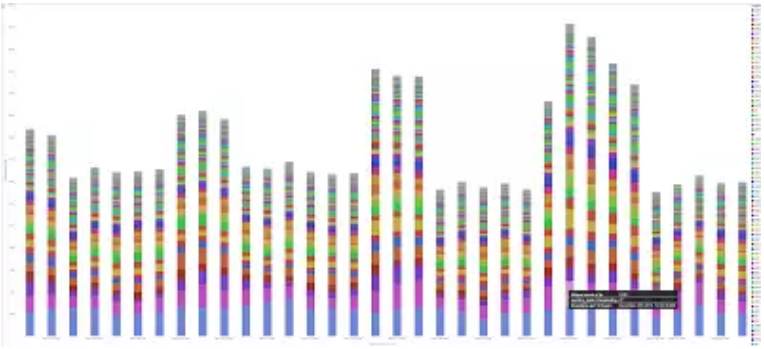
We have some early crawling numbers for this variant. But first, is may be useful to compare with the previous ZeroAccess 2 botnet. We see for ZeroAccess 2 daily distribution seemed rather consistent. The total confirmed connections were approximately 715,000 for the month.



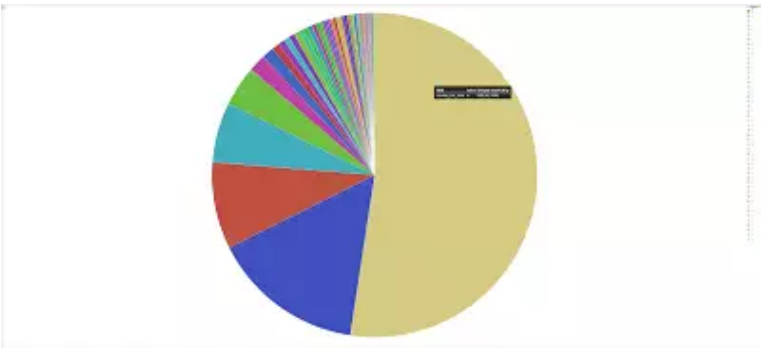
Looking deeper into a sub aggregation of country and unique IPs counted we can see a high level in US on Comcast, although this could be explained by IP churning due to DHCP for the cable provider. Nonetheless, this is a high number of uniques compared to any other ISP.



Finally, we see a rather healthy distribution of nodes and countries with a heavy weight in Japan and US. Additionally, we observe an unusual high amount of unique IPs the day after Christmas.



In comparison, the latest Zero Access 3 variant, we see the IP distribution is primarily focused in Russia. Whether this team is preparing the infrastructure for a larger planned botnet or it is focusing its efforts in the area is still unknown.



We can potentially hypothesize that this botnet has been retooled to allow botnet subgroups using the generation of new encryption keys and RSA binary signing keys. The result of this would mean multiple actors could use the same malware, similarly to trends that have been observed with other seed-based malware, like Tiny Banker (aka Tinba). This would make takedowns more difficult in that there is no one central peer to peer network, but rather a cluster of peer to peer networks. Another plausible hypothesis is that the ZeroAccess code base has been sold (or stolen) to a third party, one who is more interested in targeting Eastern Europe.

These types of networks are becoming more and more resilient to takedowns, as observed with Dridex, and even ZeroAccess itself is still maintaining its presence. With multiple authors and actors involved, the difficulty of takedown attempts by technical means is also compounded by the difficulty of prosecuting multiple authors (and actors) in multiple geopolitically diverse jurisdictions.

We would like to thank MalwareTech for his invaluable contributions to this research.