Versatile DDoS Trojan for Linux

securelist.com/versatile-ddos-trojan-for-linux/64361/

Authors

Expert Mikhail Kuzin

In February 2014, an article was published on a popular Russian IT website under a curious title – Studying the BillGates Linux Botnet. It described a Trojan with sufficiently versatile DDoS functionality. The capability that we found the most interesting was the Trojan's ability to conduct DNS Amplification-type attacks. In addition, it followed from the article that the Trojan had a sophisticated modular structure, something we had not seen in the world of Linux malware before.

The article also provided a link for downloading all of the Trojan's files (taken directly from an infected machine) – which is what we did.

The archive that we downloaded contained the following files, which, according to the author of the article, were all modules of the same Trojan:

- atddd;
- cupsdd;
- cupsddh;
- ksapdd;
- kysapdd;

- skysapdd;
- xfsdxd.

The files cupsdd and cupsddh are detected by Kaspersky Lab products as Backdoor.Linux.Ganiw.a; atddd and the remaining files are detected as Backdoor.Linux.Mayday.f.

The archive with the files also contained a configuration file for cron – the Linux task scheduler. In this case, the utility is used by the Trojan as a means of getting a foothold in the system. The Trojan uses cron to perform the following tasks:

1. Once a minute – terminate the processes of all applications that can interfere with its operation: .IptabLes, nfsd4, profild.key, nfsd, DDosl, lengchao32, b26, codelove, node24
2. Approximately once in ninety minutes – terminate all of its processes: kysapd, atdd, skysapd, xfsdx, ksapd
3. Approximately once in two hours – download all of its components to the /etc folder from http://www.dgnfd564sdf.com:8080/[module_name] (module_name = name of the Trojan's module, e.g., cupsdd), after deleting these files from the /etc folder
4. Once in ninety minutes – relaunch all of its modules
5. Every minute – purge system logs and bash command history and execute chmod 7777 [module_name]

During subsequent analysis of the files, we did not find any code responsible for saving the config file for cron. Most likely, the file was manually downloaded to the victim machine by a cybercriminal after gaining remote access to the system.

Backdoor.Linux.Mayday.f (atddd)

The file atddd is a backdoor designed to conduct various types of DDoS attacks against the servers specified. As mentioned above, Kaspersky Lab products detect it as Backdoor.Linux.Mayday.f. The files kysapdd, skysapdd, xfsdxd, ksapdd are almost exact copies of atddd – with one exception, which is discussed later in the text.

The backdoor starts its operation by calling the function daemon(1, 0), continuing to run in the background and redirecting standard input, output and errors to /dev/null

Next, atddd collects relevant information about the system, including:

1. system version (by calling uname())
2. number of CPU cores and their clock rates (taken from /proc/cpuinfo)
3. CPU load (taken from /proc/stat)
4. network load (data for interfaces with the "eth" prefix taken from /proc/net/dev)

The information listed above is stored in the g_statBase structure.

```
1 int __cdecl CStatBase::Initialize(CStatBase *this)
2 {
3   CStatBase::GetSysVersion(this);
4   CStatBase::GetCpuSpd(this);
5   CStatBase::InitGetCPUUse(this);
6   return CStatBase::InitGetNetUse(this);
7 }
```

After this, the backdoor decrypts strings defining the C&C server’s IP address and port number. The encryption algorithm used is very simple: an encrypted string is taken character-by-character, with 1 added to the ASCII code of a character if its number is odd and subtracted from it if the character’s number is even. As a result, the string “3/3-2/4-269-85” yields the IP-адрес “202.103.178.76”, while “2/:82” stands for port number “10991”.

After this, atddd reads configuration file fwke.cfg, which is located in the same folder with the malicious program. Information from the config file is saved in the g_fakeCfg structure. If the file does not exist, the backdoor attempts to create it and store the following information in it:

1st line: 0 *//flag, if 1 then begin attack, if 0 then terminate attack*

2nd line: 127.0.0.1:127.0.0.1 *//range of outgoing IP addresses*

3rd line: 10000:60000 *//outgoing port range for an attack*

4th line: an empty line *//domain name in the case of DNS flood (see below)*

This information is subsequently sent to the C&C server and can be updated with a command from the C&C.

Next, the backdoor creates a new thread, CThreadTaskManager::ProcessMain(), in which commands to begin an attack and terminate an attack are put into the execution queue. After this, a new thread is created – CThreadHostStatus::ProcessMain(). In this thread, data on CPU and network load is updated every second and can subsequently be sent to the C&C server if requested.

After this, 20 threads are created, which read information from the task queue and, depending on the information read, launch an attack or terminate it. However, some of the threads may not be used in an attack if the relevant C&C command has a parameter (the number of threads to be used).

```
CThreadTaskManager::CThreadTaskManager(a1 + 220, a1);
CThreadHostStatus::CThreadHostStatus(a1 + 276, a1);
std::allocator<CThreadAttack *>::allocator(&v5);
std::vector<CThreadAttack *,std::allocator<CThreadAttack *>>::vector(a1 + 332, &v5);
std::allocator<CThreadAttack *>::~allocator(&v5);
CThreadMutex::CThreadMutex(a1 + 344);
CTaskInfo::CTaskInfo((CTaskInfo *)(a1 + 368));
CThreadMutex::InitMutex((CThreadMutex *)(a1 + 344));
for ( i = 0; i <= 19; ++i )
{
  v1 = (CThread *)operator new(0x160u);
  CThreadAttack::CThreadAttack(v1, a1);
  v4 = v1;
  if ( v1 )
    std::vector<CThreadAttack *,std::allocator<CThreadAttack *>>::push_back(a1 + 332, &v4);
}
```

Then the malware enters an endless loop of processing messages from the C&C. After a connection with the C&C is established, information about system version and CPU clock rate, as well as data from the g_fakeCfg structure, is sent to the C&C every 30 seconds.

In response, the server should send 4 bytes, the first of which is the serial number of a command – from 1 to 4.

```
*(_DWORD *)&v53[3] = CNetBase::Recv((CNetBase *)&v51, *(CNetBase **)this, (int)&v40, (void *)4, 0xEA60u, v24);
if ( *(_DWORD *)&v53[3] != 4 && v51 != -11 )
{
  CManager::DestroySocket(this);
  goto LABEL_53;
}
if ( v51 == -11 || v51 == -11 )
{
  CManager::DestroySocket(this);
  goto LABEL_53;
}
if ( v40 != 2 && v40 != 1 && v40 != 3 && v40 != 4 )
{
  CManager::DestroySocket(this);
```

Next, if the command has parameters, the C&C sends another 4 bytes defining the amount of data that will be sent (i.e., the parameters). Then the parameters themselves are sent; their size should match the number from the previous C&C response.

About each command in greater detail:

- 0x01. Command to launch an attack. Parameters define the attack's type and the number of threads to be used. The attack type is defined by a byte which can take values from 0x80 to 0x84. This means that 5 attack types are possible:
  - 0x80 – TCP flood. The destination port number is sent by the C&C in its response as a parameter. The source port range is defined in fwke.cfg. Each new request is sent from a new port within the range defined, in the ascending order of port numbers. The destination IP address is also defined in parameters.
  - 0x81 – UDP flood. The same as 0x80, but UDP is used as the transport layer protocol.
  - 0x82 – ICMP flood. Same as above, but via ICMP.

- 0x83, 0x84 – two DNS flood attacks. The only difference is the domain name sent in the DNS request. In the former case, the name is generated randomly, in the second, it is defined in a parameter (the fourth line in fwke.cfg). Essentially, both attacks are similar to 0x81, except that port 53 (the default port for the DNS service) is used as destination port.
- 0x02. Command to terminate an attack. The value in the first line of fwke.cfg is changed to 0 and the attack is terminated.
- 0x03. Command to update the file fwke.cfg. The response also includes a structure similar to g_fakeCfg, data from which is used to create the new fwke.cfg file.
- 0x04. Command to send the current command's execution status to the C&C server.

In addition to the above, the backdoor includes several empty methods (without any code), which have curious names: CThreadAttack::EmptyConnectionAtk, CThreadAttack::FakeUserAtk, CThreadAttack::HttpAtk. Apparently, the malware writer had plans to extend the malicious program's functionality, and this is a test version rather than a final version. The file cupsdd, which is discussed below, provides a confirmation of this.

The files kysapdd, skysapdd, xfsdxd, ksapdd are almost identical copies of atddd, with the exception that they contain different C&C server addresses: 112.90.252.76:10991, 112.90.22.197:10991, 116.10.189.246:10991 and 121.12.110.96:10991, respectively. The names of their configuration files are also different: fsfe.cfg, btgw.cfg, fake.cfg, xcke.cfg, respectively.

This means that, contrary to our expectations, the files atddd, kysapdd, skysapdd, xfsdxd, ksapdd are not modules of a single piece of malware but rather different copies of the same Trojan, each connecting to its own C&C server. However, this is not the most curious part of it by far.

Backdoor.Linux.Ganiw.a (cupsdd)

Like the files described above, this file is a backdoor designed to carry out various types of DDoS attacks. However, cupsdd is significantly more feature-rich and sophisticated than its 'colleagues', although in places its code is very similar to that found in atddd.

The backdoor starts its operation by initializing the variables it needs from the string "116.10.189.246:30000:1:1:h:578856:579372:579888" (separator – ":"), which it first decrypts using the RSA algorithm. The string contains the following variables:

g_strConnTgt=116.10.189.246 – C&C server's IP address

g_iGatsPort=30000 – C&C server's port

g_iGatsIsFx=1 and g_iIsService=1 – flags used later

g_strBillTail=h – postfix for the name of the file that will be dropped (see below)

g_strCryptStart=578856, g_strDStart=579372, g_strNStart=579888 – pointers to RSA data (encrypted string and key)

Next, the malware drops and executes a file, which is located at offset 0xb1728 from the beginning of the original file and is 335872 bytes in size, provided that the file is not already running. The malware checks whether the file is running by trying to bind a socket to 127.0.0.1:10808. If the attempt is successful, it means that the file is not running, and it needs to be dropped and executed.

```
signed int CSysTool::IsBillExist()
{
  unsigned __int16 v0; // ax@3
  signed int v2; // [sp+4h] [bp-14h]@4
  unsigned __int32 args; // [sp+10h] [bp-8h]@1

  args = CNetBase::CreateSocket(6);
  if ( (args & 0x80000000) != 0 )
    exit(0);
  v0 = atoi("10808");
  if ( CNetBase::Bind(args, v0) )
  {
    CNetBase::Close(args);
    v2 = 1;
  }
```

If the file is already running, its process, whose PID can be found in the file /tmp/bill.lock, is terminated (kill(pid, 9)). Then the file is dropped anyway, replacing the existing copy.

The name of the file that is dropped is generated from the name of the current file that is running + postfix from the variable g_strBillTail. In our case, the file was named cupsddh and was located in the same folder with the dropper.

After this, the current process forks and the child process calls the function system("/path/to/cupsddh"), which executes the file dropped.

Next, the function daemon(1, 0) is called, for the same purpose as in the case of the sample described above (atddd).

After this, the malware handles the situation if cupsdd was executed earlier and is currently active. For this purpose, it checks whether the file /tmp/gates.lock exists. If it does exist, the current process is terminated (exit(0)). If not, the file (/tmp/gates.lock) is created and the PID of the current process is written to it.

Then, if flag g_iIsService == 1, the backdoor sets itself to run at system startup by creating the following script named DbSecuritySpt in /etc/init.d/:

#!/bin/bash
/path/to/cupsdd

The malware also creates symbolic links to the script in /etc/rc[1-5].1/S97DbSecuritySpt

```
if ( g_iIsService == 1 )
  CUtility::SetAutoStart((int)"DbSecuritySpt", 97);
```

```
CUtility::Sleep(1000);
std::allocator<char>::allocator();
std::string::string((int)&v11, (int)"/etc/init.d/", (int)&v12);
std::operator+<char,std::char_traits<char>,std::allocator<char>>((int)&v9, (int)&v11, a1);
std::string::~string(&v11);
std::allocator<char>::~allocator();
v2 = std::string::c_str((int)&v9);
fd = open((char *)v2, 578, 0x1EDu);
if ( fd > 0 )
{
  memcpy(&addr, &CUtility::SetAutoStart(char const*,int)::C.63, 256);
  v3 = std::string::c_str((int)&v10);
  sprintf(&addr, "#!/bin/bash\n%s\n", v3);
  write(fd, &addr, strlen(&addr));
  close(fd);
  for ( i = 0; i <= 4; ++i )
  {
    memcpy(&v7, &CUtility::SetAutoStart(char const*,int)::C.64, 256);
    memcpy(&pathname, &CUtility::SetAutoStart(char const*,int)::C.65, 256);
    sprintf(&pathname, "/etc/rc%d.d/S%d%s", i + 1, a2, a1);
    if ( access(&pathname, 0) != 0 )
    {
      sprintf(&v7, "ln -s /etc/init.d/%s %s", a1, &pathname);
      system(&v7);
    }
  }
  v5 = 1;
}
else
{
  v5 = 0;
}
std::string::~string(&v9);
```

Next, the malware reads the configuration file conf.n (if it exists) from the same folder as the one in which cupsdd is located. The first 4 bytes of the file define the size of the data which follows. All the data is stored in the structure g_cnfgDoing.

Then the malware reads the file containing commands – cmd.n. The format is the same as in conf.n. The data is stored in the structure g_cmdDoing.

After this, the malware obtains all the necessary information about the system, including:

- The operating system's name and kernel version (e.g., Linux 3.11.0-15-generic), by calling uname()
- CPU clock rate, taken from /proc/cpuinfo
- Number of CPU cores, taken from /proc/cpuinfo, and CPU load, taken from /proc/stat
- Network load, taken from /proc/net/dev
- Hard drive size in megabytes, taken from /proc/meminfo
- Information about network interfaces, taken from /proc/net/dev

All the data is stored in the structure g_statBase.

Next, a new stream, CThreadTaskGates::ProcessMain, is created, in which the following commands are processed:

- 0x03. DoConfigCommand(). Update the configuration file conf.n.
- 0x05. DoUpdateCommand(). Start a new thread, CThreadUpdate::ProcessMain, in which update one of its components. The command accepts a number from 1 to 3 as a parameter. Each of the numbers is associated with one of the following strings:
  1 – “Alib” – the file /usr/lib/libamplify.so
  2 – “Bill” – the dropped module (cupsddh)
  3 – “Gates” – the dropper (cupsdd)

```
loc_8057AD6:                            ; CODE XREF: CManager::UpdateProcess(C
                sub     esp, 8
                push    offset aBill    ; "Bill"
                lea     eax, [ebp+var_20]
                push    eax
                call    _ZNSsaSEPKc     ; std::string::operator=(char const*)
                add     esp, 10h
                jmp     short loc_8057B35
; ---------------------------------------------------------------------------

loc_8057AEC:                            ; CODE XREF: CManager::UpdateProcess(C
                sub     esp, 8
                push    offset aAlib    ; "Alib"
                lea     eax, [ebp+var_20]
                push    eax
                call    _ZNSsaSEPKc     ; std::string::operator=(char const*)
                add     esp, 10h
                jmp     short loc_8057B35
; ---------------------------------------------------------------------------

loc_8057B02:                            ; CODE XREF: CManager::UpdateProcess(C
                sub     esp, 8
                push    offset aGates   ; "Gates"
                lea     eax, [ebp+var_20]
                push    eax
                call    _ZNSsaSEPKc     ; std::string::operator=(char const*)
                add     esp, 10h
                mov     [ebp+var_11], 1
                jmp     short loc_8057B35
```

Depending on the parameter, one of the malicious program’s components is updated. An update is launched by sending 6 bytes containing the string “EF76#^” to the C&C server, followed by one of the strings described above (depending on the parameter).

The C&C responds by sending 4 bytes containing the length (in bytes) of the file that will be transferred next. Then the C&C transfers the file itself in 1024-byte packets.

First, the file is saved in the /tmp folder under a random name consisting of digits. Then, depending on the file that was received, the existing file cupsdd (or cupsddh) is replaced or the file is copied to /usr/lib/libamplify.so

Next, the temporary file is deleted from /tmp, and the chmod command is used to set 755 permissions for the resulting file. After this, in the case of updating cupsddh, the active process is terminated and the new file is launched. In the case of updating cupsdd, the final stage (starting from copying a file from /tmp) is carried out by cupsddh, to which the relevant command is sent.

- 0x07. DoCommandCommand(). Write a new command to cmd.n.
- 0x02. StopUpdate(). Close the current connection, which was established in order to update modules.

Next, the backdoor starts several threads, in which it simultaneously performs several additional operations:

- CThreadClientStatus updates the data on CPU and network load in the g_statBase structure every second.
- CThreadRecycle removes completed tasks from the queue.
- CThreadConnSender reads commands from the queue and passes them to the cupsddh module via a TCP connection with 127.0.0.1 on port 10808. In response it receives the status of their execution.
- CThreadMonBill checks whether the module cupsddh is running once every minute. If not, it drops and executes it again.
- CThreadLoopCmd reads commands from g_cmdDoing (the file cmd.n) and executes them using the call system(cmd).

Next, the main thread enters the loop of receiving and processing commands from the C&C server. There are two possibilities in this case, depending on the g_iGatsIsFx flag:

1. If the flag is set (==1), the malware simply uses the new thread to send information about the system and the current configuration from g_cnfgDoing to the C&C and waits to receive commands in response, like the sample (atddd) described above;
2. If the flag is not set, then the communication session is initiated by the C&C. In other words, the malware waits for the C&C to connect and begins to transfer the data mentioned above only when a connection has been established.

```
if ( g_iGatsIsFx )
{
  for ( i = 0; ; ++i )
  {
    v5 = i;
    if ( (unsigned int)v5 >= std::vector<std::string,std::allocator<std::string>>::size(a1) )
      break;
    v2 = std::vector<std::string,std::allocator<std::string>>::operator[](a1, i);
    v3 = std::string::c_str(v2);
    v4 = operator new(64);
    CThreadFXConnection::CThreadFXConnection(v4, a1, v3);
    v10 = v4;
    if ( v4 )
    {
      CThread::StartThread(v10);
      CAutoLock::CAutoLock((int)&v7, a1 + 844, 1);
      v8 = v10;
      std::set<void *,std::less<void *>,std::allocator<void *>>::insert(&v6);
      CAutoLock::~CAutoLock(&v7);
    }
  }
}
else
{
  CManager::ZXMainProcess(esi0, a1);
}
```

Commands received from the C&C are divided into two queues: either for execution in the current module (in the thread CThreadTaskGates described above) or for passing to the cupsddh module (thread CThreadConnSender).

Backdoor.Linux.Ganiw.a (cupsddh)

The file is packed with UPX. After being unpacked, it calls daemon(1,0). It creates the file /tmp/bill.lock, in which it stores the PID of the current process. cupsddh stores system data in the structure g_statBase, which is identical to that used by cupsdd.

Next, it populates the structure g_provinceDns with the IP addresses of DNS servers converted to binary data in network byte order using the function inet_addr(), taking data from the string array g_sProvinceDns (offset in unpacked file: 0x8f44c, size 4608 bytes).

cupsddh executes command “insmod /usr/lib/xpacket.ko” in an attempt to load the kernel module into the kernel. However, the file is not present on ‘clean’ systems, and the malware does not make any attempt to download it or obtain it in any other way.

```
void __noreturn MainProcess()
{
  int v0; // [sp+10h] [bp-18h]@1

  MarkBill();
  CStatBase::Initialize((int)&g_statBase);
  CProvinceDns::InitReadResource((int)&g_provinceDns);
  system((int)"insmod /usr/lib/xpacket.ko");
  CAmpResource::InitReadResource((int)&g_AmpResource, "/usr/lib/libamplify.so");
  v0 = operator new(580);
  CManager::CManager(v0);
  if ( v0 )
    CManager::Initialize(v0);
```

Next, data from the file /usr/libamplify.so (as it turns out, this is not a library but one more config file) is loaded into the structure g_AmpResource. The file has the following format: 1st dword is the number of dwords that follow. Apparently, it contains the list of IP addresses for DNS servers that are currently relevant, i.e., those suitable for DNS Amplification type DDoS attacks.

After this, the module creates two threads: CThreadTask and CThreadRecycle. The former executes commands from a queue comprising commands which came from the cupsdd module. The latter removes commands that have been executed. Next, the main thread binds a socket to 127.0.0.1:10808 and enters an endless loop, receiving commands from the cupsdd module and putting the commands received into the above queue.

The following commands are possible:

- 0x01. Start an attack according to the parameters received. See a more detailed description below.
- 0x02. Terminate the current attack, setting the relevant flag.
- 0x03. Update the current configuration in the g_cnfgDoing structure, which is used during an attack. Also, update the current local MAC address, as well as the MAC and IP address of the current gateway in the structure g_statBase.
- 0x05. The final stage of updating the cupsdd module (described above).

```
v11 = CNetBase::Recv((int)&v7, args, (int)&v9, 4, 20000);
if ( v11 == 4 )
{
  v11 = CNetBase::Recv((int)&v7, args, (int)&v8, 4, 20000);
  if ( v11 == 4 )
  {
    std::allocator<char>::allocator(&v10);
    std::vector<char,std::allocator<char>>::vector(&v6, &v10);
    std::allocator<char>::~allocator(&v10);
    if ( v8 <= 0
      || (std::vector<char,std::allocator<char>>::resize(&v6, v8),
         v2 = v8,
         v3 = std::vector<char,std::allocator<char>>::operator[](&v6, 0),
         v11 = CNetBase::Recv((int)&v7, args, v3, v2, 20000),
         v11 == v8) )
    {
      if ( (unsigned int)v9 <= 5 )
        JUMPOUT(__CS__, *(&command_table_ptr + v9));
```

Two main attack modes are possible: normal mode and kernel mode.

Kernel mode

This mode uses pktgen, a kernel-level packet generator built into Linux. Its advantage to the attacker is that traffic is generated with the greatest speed possible for the given network interface. In addition, the packets generated in this way cannot be detected using ordinary sniffers, e.g., the standard tcpdump, since packets are generated at kernel level.

```
for ( i = 0; ; ++i )
{
  v2 = *(_DWORD *)(a1 + 156) <= i || CStatBase::CpuNum((int)&g_statBase) <= i ? 0 : 1;
  if ( !v2 )
    break;
  CThreadKernelAtkExcutor::KCfgDev(a1, i);
}
for ( j = 0; ; ++j )
{
  v3 = *(_DWORD *)(a1 + 156) <= j || CStatBase::CpuNum((int)&g_statBase) <= j ? 0 : 1;
  if ( !v3 )
    break;
  CThreadKernelAtkExcutor::KCfgCfg(a1, j);
}
result = CThreadKernelAtkExcutor::KWriteCmd(a1, "/proc/net/pktgen/pgctrl", (mode_t)"start");
```

The packet generator is controlled using a set of scripts/configs in the /proc/net/pktgen folder, but the module pktgen must first be loaded into the kernel by calling the command “modprobe pktgen”. However, I did not find any such calls. Apparently, the call “insmod /usr/lib/xpacket.ko” is used instead, although, as mentioned above, the file is absent from the system by default. As a result, kernel mode is not operational in this version of the malware.

Nevertheless, the malware attempts to write several files to the /proc/net/pktgen folder, namely:

1. the file /proc/net/pktgen/kpktgend_%d for each CPU core, where %d is the core number, beginning from 0. The file's contents is as follows:
   rem_device_all
   add_device eth%d
   max_before_softirq 10000

```
sprintf(v2, "/proc/net/pktgen/kpktgend_%d", a2);
if ( a2 >= 0 )
{
  v4 = (char *)std::vector<char,std::allocator<char>>::operator[](&v8, 0);
  CThreadKernelAtkExcutor::KWriteCmd(a1, v4, (mode_t)"rem_device_all");
  v5 = (char *)std::vector<char,std::allocator<char>>::operator[](&v8, 0);
  CThreadKernelAtkExcutor::KWriteCmd(a1, v5, (mode_t)"add_device eth%d", a2);
  v6 = (char *)std::vector<char,std::allocator<char>>::operator[](&v8, 0);
  CThreadKernelAtkExcutor::KWriteCmd(a1, v6, (mode_t)"max_before_softirq 10000");
}
else
{
  v3 = (char *)std::vector<char,std::allocator<char>>::operator[](&v8, 0);
  CThreadKernelAtkExcutor::KWriteCmd(a1, v3, (mode_t)"rem_device_all");
}
```

2. the file /proc/net/pktgen/eth%d for each CPU core, where %d is the core number, beginning from 0. The file's contents is as follows:
   count 0
   clone_skb 0
   delay 0
   TXSIZE_RND
   min_pkt_size %d
   max_pkt_size %d
   IPSRC_RND
   src_min %s
   src_max %s
   UDPSRC_RND
   udp_src_min %d
   udp_src_max %d
   dst %s
   udp_dst_min %d
   udp_dst_max %d
   dst_mac %02x:%02x:%02x:%02x:%02x:%02x *//MAC address of the gateway from g_statBase*
   is_multi %d
   multi_dst %s *//if there are several addresses to be used in an attack (i.e., if the value in the previous line is not equal to 0), they are set in these lines, the number of which matches the previous parameter*
   pkt_type %d
   dns_domain %s
   syn_flag %d
   is_dns_random %d
   dns_type %d
   is_edns %d
   edns_len %d
   is_edns_sec %d

   The values of most pktgen parameters are passed from cupsdd via command parameters.

3. the file /proc/net/pktgen/pgctrl, which contains the string "start".

Normal attack mode

As in the case of atddd, normal attack mode uses raw sockets.

The following attack types are possible in this mode:

- CAttackSyn – TCP-SYN flood.

- CAttackUdp – UDP flood (as in the case of atddd)
- CAttackDns – DNS flood (as in the case of atddd)
- CAttackIcmp – ICMP flood (as in the case of atddd)
- CAttackCc – HTTP flood.
- CAttackAmp – DNS Amplification.

The last attack type on the list above is different in that packets are sent to vulnerable DNS servers, with the attack target specified as the sender's IP address. As a result, the cybercriminal sends a small packet with a DNS request and the DNS server responds to the attack target with a significantly larger packet. The list of vulnerable DNS servers is stored in the file libamplify.so, which is written to disk following the relevant command from the C&C.

```
int __cdecl CThreadNormalAtkExcutor::ProcessMain(int a1)
{
  int result; // eax@3
  void (__cdecl *v2)(_DWORD, _DWORD); // ebx@17
  int v3; // eax@17
  int v4; // [sp+Ch] [bp-3Ch]@1
  signed int v5; // [sp+3Ch] [bp-Ch]@8

  v4 = *(_BYTE *)(a1 + 128) - 16;
  if ( (unsigned int)v4 <= 49 )
    JUMPOUT(__CS__, *(&normalAtkTypeTable + v4));
```

Post Scriptum. BillGates v1.5

This version of the Trojan appeared a little later and is probably currently the latest. Essentially, this is the same cupsdd, only a little 'shaped up'. Overall, there is more logic in the code, plus there are a couple of new functions.

The most significant changes were made to the Gates module, i.e., the file cupsdd. Now it has three operating modes. The choice of mode is based on the folder from which the file is launched. Specifically, if it is launched from /usr/bin/pojie, then monitoring mode is enabled, otherwise the module operates in installation and updating mode, which is later superseded by the mode in which it controls the Bill module.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  void *v3; // esp@1
  int v5; // [sp+4h] [bp-4h]@1

  v3 = alloca(16);
  CSysTool::SelfInit();
  v5 = CSysTool::CheckGatesType();
  if ( v5 )
  {
    if ( v5 == 1 )
      MainBeikong();
  }
  else
  {
    MainMonitor();
  }
  return 0;
}
```

1. **Installation and updating mode.**
   First, the module terminates its process working in monitoring mode, if it exists. Its PID is kept in the file /tmp/moni.lock.
   Next, it reinstalls and re-launches the Bill module.
   Next, if a process working in the 'controlling the Bill module' mode exists, that process is terminated. Its PID is kept in the file /tmp/gates.lock.
   If the flag g_iIsService is set (it is defined in the same way as in the previous version), the module sets itself to run at system startup in the same way as before (in the previous version).
   Next, the module writes its path to the file /tmp/notify.file and then copies itself to the file /usr/bin/pojie. After this, it launches its copy, which is, obviously, set to run in monitoring mode, and then changes its own operating mode to controlling the Bill module.

2. **Monitoring mode.**
   Writes the PID of the current process to the file /tmp/moni.lock. Next, it starts two threads – one to monitor the Bill module and the other to monitor the Gates module operating in controlling Bill mode. If one of these processes is currently not running, the relevant file is created and launched again.

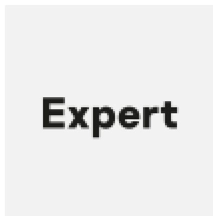
3. **Controlling the Bill module mode.**
   The actions of the Gates module are exactly the same as they were in the previous version of the Trojan (after installing the Bill module and initializing the relevant variables and structures of the Trojan).

To summarize, in the new version of the Trojan its authors have added a little 'robustness' without making any significant functionality changes.

It is also worth noting that the hard-coded IP address of the C&C server has remained the same (116.10.189.246) in this version, but the port number has changed – it is now 36008 instead of 30000 in the previous version.

- Backdoor
- Botnets
- DDoS-attacks
- DNS Amplification
- Linux

Authors



Mikhail Kuzin

Versatile DDoS Trojan for Linux

Your email address will not be published. Required fields are marked *