Pouring Acid Rain

Register Now Learn More

Blogs

The latest cybersecurity trends, best practices, security vulnerabilities, and more

By Max Kersten · April 30, 2024

In two recent major geopolitical conflicts, in Ukraine and in Israel, wipers - malware used to destroy access to files and commonly used to halt telecom operations - were used to destroy digital infrastructure. Their ongoing shows that wipers have their place in a nation's cyber arsenal, are here to stay and not a one-off tactic. Reports on the latest wipers have been making the rounds from SentinelOne regarding AcidRain and AcidPour, as well as Ruben Santamarta's confirmation of the impact of AcidRain on Viasat KA-SAT devices.

While AcidRain predates AcidPour by more than two years, the two wipers look alike in more than just their behavior. This blog will provide background information on wipers in general, explain the inner workings of both Acid wipers, compare the capabilities of both wipers, and discuss potential attribution

Background information

The date on which the AcidRain attack occurred coincides with the start of kinetic activities in the Russo-Ukrainian conflict, where wipers were likely used to gain a battlefield advantage by disrupting satellite communications, bricking the aforementioned Viasat KA-

SAT modems. These satellite-based modems allow one to move around while maintaining an internet connection, regardless of the presence (and/or condition) of cell phone towers.

The trend of using wipers within international conflicts is not new, with ample examples going back years. In 2017, we saw both WannaCry and NotPetya, wipers disguised as ransomware with no intention to be used as such, target Ukraine. Going back to 2012, the Shamoon wiper wreaked havoc on Saudi Arabia's Aramco.

Notable for all three wipers is the ability to spread, where the malware infected other systems in the network. WannaCry and NotPetya used the EternalBlue exploit, which originated from the Vault7 leak, allowing them to spread like wildfire on networks with unpatched machines. Shamoon copied itself to other machines and shares where possible, confining the spreading within the targeted network, rather than impacting companies on a global scale.

Looking back at more recent history, wiper usage has increased, especially when linked to activism and armed conflicts. In early 2022, there was an uptick of wipers used on targets within Ukraine. As noted in our Wipermania blog from November 2022, the WhisperGate and Hermetic wipers focused on that area as well, albeit prior to AcidRain. Post AcidRain, there were numerous wipers that followed: IsaacWiper, CaddyWiper I, DoubleZero, Industroyer2, Shred wipers,CaddyWiper II, RansomBoggs, SwiftSlicer, CaddyWiper III, ZeroWipe, and BidSwipe.

These wipers, written in a multitude of programming languages with different styles and targets, indicate the loss of data and loss of system uptime is a target during conflicts, even if that were to mean covert access would be burned by destroying the targeted systems.

Even more recently, wipers have been used several times in Israel, since the 7th of October 2023, which marked the start of Israel's military extensive response to Hamas. In late October 2023, Security Joes and ESET discovered two versions of the BiBi wiper (the name is a reference to Israel's prime minister Benjamin Netanyahu), after which the MultiLayer, PartialWasher, and BFG Agonizer wipers were discovered by Palo Alto Networks' Unit42, also targeting Israel. In February 2024, Intezer's Nicole Fishbein discoveredthe SameCoin wiper, about which HarfangLab published an extensive analysis.

Note that the lack of wiper malware which aims to target devices within Gaza is likely due to prioritization. Once a conflict becomes fully kinetic, defending IT infrastructure against wipers becomes of secondary importance. From the other perspective, destroying cell towers and similar infrastructure causes downtime in communication on the battlefield without the need for malware. Alternatively, disconnecting an area from the internet and/or electricity results in the same, albeit in a reversible manner.

AcidRain

AcidRain	
MD-5	ecbe1b1e30a1f4bffaf1d374014c877f
SHA-1	86906b140b019fdedaaba73948d0c8f96a6b1b42
SHA-256	9b4dfaca873961174ba935fddaf696145afe7bbf5734509f95feb54f3584fd9a
Detection names	Trojan.GenericKD.48777542

AcidRain was first submitted to VirusTotal on the 15th of March 2022, after which numerous AV, EDR, and XDR suites detected it, including Trellix' products.

To avoid repeating existing research, a high level overview of the AcidRain follows, along with some interesting decompiled code excerpts to illustrate the most important points. Below, the wiper's flowchart is given, highlighting the taken steps.

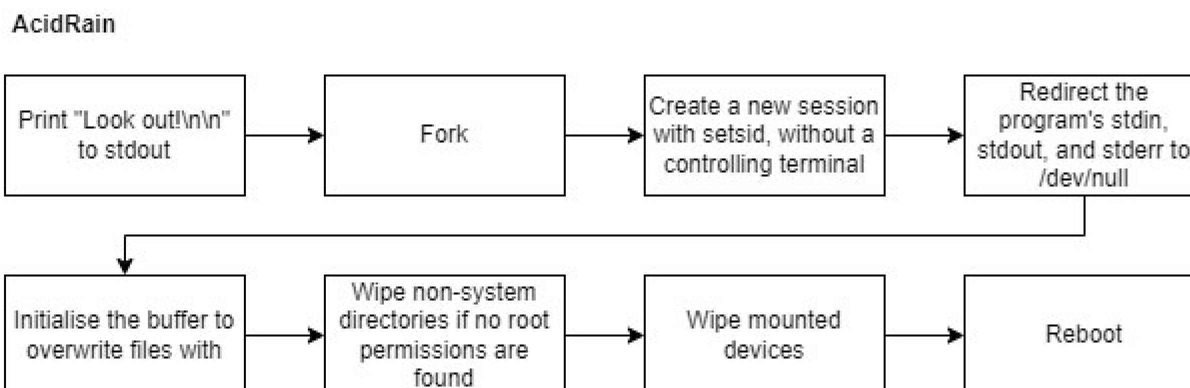

Figure 1 - AcidRain's flowchart

While some of the steps require no further explanation, such as the print statement at the beginning, it is clear that the wiping process within the wiper requires more details. The check to see if the process is running with root privilege, along with the subsequent call to wipe non-system directories, is shown below.

```
Decompile: main -  (acid_rain)
29    uid = getuid();
30    if (uid != ROOT) {
31      deleteNonSystemDirectories();
32    }
```

Figure 2 - The user ID check in AcidRain

The "getuid" (short for "get user ID") call returns the real user ID of the calling process, which is the wiper. If this is not equal to 0, which is generally the root user's ID, it calls the wipe function.

Directly after that, the execution continues by calling several functions in a row, each of which has different targets to wipe. If the current user is the root user, the non-system directories (and files therein) are then deleted. This is the same function as the one mentioned above, indicating that the deletion of these directories is certain, but the moment depends on the user which executes the binary.

```
Decompile: main -  (acid_rain)
33    wipeDevicesCharacters();
34    wipeByFlashingMtdBlocks();
35    wipeMMCBlocks();
36    wipeMtdDevices();
37    wipeLoopDevices();
38    uid_ = getuid();
39    if (uid_ == ROOT) {
40      deleteNonSystemDirectories();
41    }
```

Figure 3 - The wipe-related functions in AcidRain

The affected directories are listed in the table below, along with a brief description of the directory's purpose.

Directory	Purpose
/dev/sd	Contains the Small Computer System Interface (SCSI) disks in the order as they are discovered by the operating system
/dev/mtdblocks, /dev/mtd, and /dev/block/mtdblocks	Memory Technology Device (MTD) allows the operating system to deal with flash memory
/dev/mmcblk and /dev/block/mmcblk	The eMMC flash storage block devices available on the device
/dev/loop	Loop devices are used to mount regular files as block devices, which can then be accessed normally

Once the wiping is complete, the device is rebooted.

AcidPour

AcidPour	
MD-5	1bde1e4ecc8a85cffef1cd4e5379aa44
SHA-1	b5de486086eb2579097c141199d13b0838e7b631

SHA-256	6a8824048417abe156a16455b8e29170f8347312894fde2aabe644c4995d7728
Detection names	Trojan.Linux.GenericKD.24993

AcidPour was first submitted to VirusTotal on the 16th of March 2024, after which numerous AV, EDR, and XDR suites detected it, including Trellix' products. Much like AcidRain's section, this section will provide a high level overview of the AcidPour wiper, along with some examples and interesting tidbits. The flowchart below showcases the wiper's execution flow.

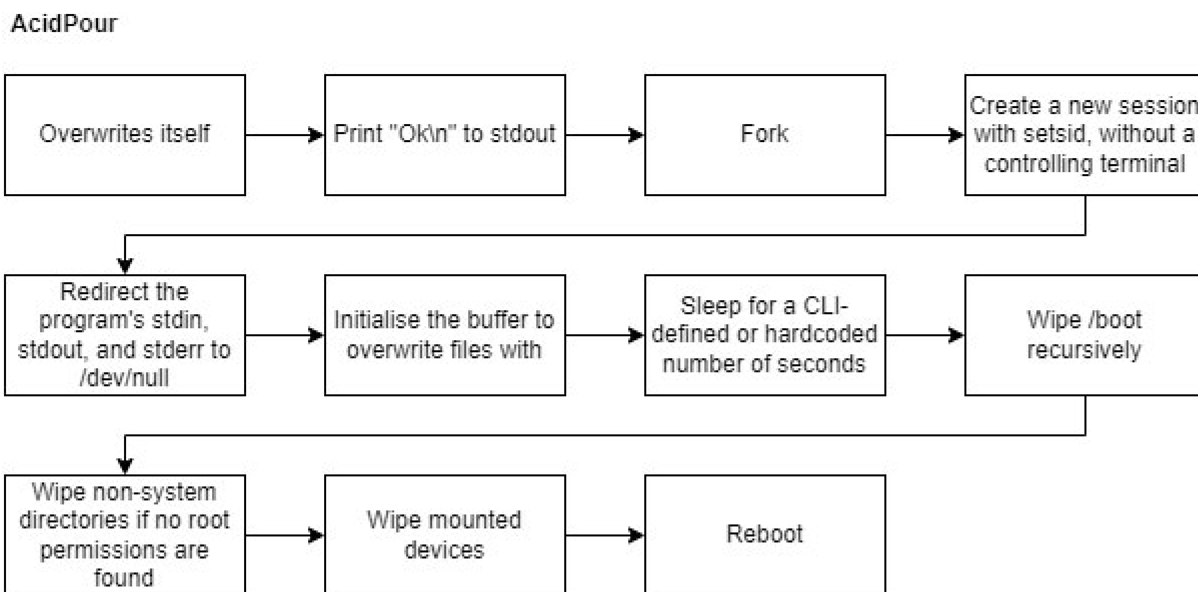

Figure 4 - The AcidPour flowchart

One of the new capabilities is the wiper's ability to overwrite itself. Usually, self deletion of malware is done to avoid the sample being found during the incident response and subsequently being analyzed. On Linux based systems, files which are in-use can be overwritten, contrary to Windows. In the image below, the link to "/proc/self/exe" is read, which links to the binary of the current executable.

```
Decompile: overwriteSelf -  (acid_pour)
15   stringLength = readlink("/proc/self/exe",pathToSelf,0x1ff);
16   if (stringLength - 1U < 0x1fe) {
17     pathToSelf[stringLength] = '\0';
18   }
19   else {
20     pathToSelf[0] = '/';
21     pathToSelf[1] = 'p';
22     pathToSelf[2] = 'r';
23     pathToSelf[3] = 'o';
24     pathToSelf[4] = 'c';
25     pathToSelf[5] = '/';
26     pathToSelf[6] = 's';
27     pathToSelf[7] = 'e';
28     pathToSelf[8] = 'l';
29     pathToSelf[9] = 'f';
30     pathToSelf[10] = '/';
31     pathToSelf[11] = 'e';
32     pathToSelf[12] = 'x';
33     pathToSelf[13] = 'e';
34   }
```

Figure 5 - Getting the path to the wiper executable

It then obtains a file descriptor to said binary, as can be seen on line 40 in the image below. It then creates a buffer, where each entry into the array is an increment of the previous value, with 1023 being the highest value. This buffer is then iteratively written over the original binary.

```
Decompile: overwriteSelf - (acid_pour)
35   statResult = stat(pathToSelf,&statStruct);
36   if (statResult != SUCCESS) {
37     statStruct.st_size = 1000000;
38   }
39   unlink(pathToSelf);
40   fdSelf = open(pathToSelf,O_WRONLY | O_PATH | O_NOATIME,RWX);
41   if (-1 < fdSelf) {
42                     /* Fill buffer, starting at 0, incrementing by one per iteration */
43     i = 0;
44     do {
45       buffer[i] = (byte)i;
46       i = i + 1;
47     } while (i != 0x400);
48     if (statStruct.st_size != -0x401 && -1 < statStruct.st_size + 0x401) {
49       totalWrittenBytes = 0;
50       do {
51         writtenBytes = write(fdSelf,buffer,0x400);
52                     /* If a zero or negative number of bytes is written, the call failed */
53         if (writtenBytes < 1) break;
54         totalWrittenBytes = totalWrittenBytes + writtenBytes;
55       } while (totalWrittenBytes < statStruct.st_size + 0x401);
56     }
57     fsync(fdSelf);
58     fsync(fdSelf);
59     close(fdSelf);
60   }
61   return;
62 }
```

Figure 6 - Overwriting the binary with an ascending buffer

The second addition to the wiper is the delay prior to the start of the wiping. The duration, in seconds, of this delay can be specified via the first command-line interface argument. If no such argument is provided, a delay of 613 seconds is applied. The image below shows the relevant decompiled code from the wiper.

```
Decompile: main - (acid_pour)
38    if (argc < 2) {
39      sleep(0x265);
40    }
41    else {
42      secondsToSleep = convertStringToInteger(argv[1]);
43      i = 0;
44      do {
45        argvCopy = argv[i];
46        pArgv = *argvCopy;
47        while (pArgv != '\0') {
48          *argvCopy = '\0';
49          argvCopy = argvCopy + 1;
50          pArgv = *argvCopy;
51        }
52        i = i + 1;
53      } while (argc != i);
54      sleep(secondsToSleep);
55    }
```

Figure 7 - The sleep-logic prior to wiping

The third addition is the recursive wiping of “/boot”. The decompiled code at line 24 in the image below contains the call to read the given directory’s content. Lines 52 and 55 contain checks with regards to the specific path. If it is a regular file (“DT_REG”) or a symbolic link (“DT_LNK”), the file is wiped. If it is a directory (“DT_DIR”), the recursive wipe function is called with the newly discovered directory as its argument, thus traversing all directories within each directory, starting at the provided directory. At last, the directory is removed once it is empty. Lastly, the unlink function is called to remove the specific file. Note that “buffer” is a copy of the provided input path, which is copied prior to the code shown in the figure.

```
Decompile: recursivelyWipe -  (acid_pour)
21  hDir = opendir(input);
22  if (hDir != NULL) {
23    while( true ) {
24      pDirectoryEntry = readdir(hDir);
25      result = pDirectoryEntry == NULL;
26      if (result) break;
27      directoryName = pDirectoryEntry->d_name;
28      iVar1 = 2;
29      pcVar2 = directoryName;
30      pcVar3 = ".";
31      do {
32        if (iVar1 == 0) break;
33        iVar1 = iVar1 + -1;
34        result = *pcVar2 == *pcVar3;
35        pcVar2 = pcVar2 + 1;
36        pcVar3 = pcVar3 + 1;
37      } while (result);
38      if (!result) {
39        iVar1 = 3;
40        pcVar2 = directoryName;
41        pcVar3 = "..";
42        do {
43          if (iVar1 == 0) break;
44          iVar1 = iVar1 + -1;
45          result = *pcVar2 == *pcVar3;
46          pcVar2 = pcVar2 + 1;
47          pcVar3 = pcVar3 + 1;
48        } while (result);
49        if (!result) {
50          strncpy(buffer + inputLength + 1,directoryName,0x1fe - (inputLength + 1));
51          directoryType = pDirectoryEntry->d_type;
52          if ((directoryType == DT_REG) || (directoryType == DT_LNK)) {
53            wipe(buffer);
54          }
55          else if (directoryType == DT_DIR) {
56            recursivelyWipe(buffer);
57            rmdir(buffer);
58          }
59          unlink(buffer);
60        }
61      }
```

Figure 8 - The recursive wipe function

Much like AcidRain, AcidPour wipes devices. Aside from the affected directories mentioned at AcidRain, two more directories, explained below, have been added.

Directory	Purpose

/dev/ubi	Unsorted block images (UBI), used in flash file systems
/dev/dm-	Logical Volume device mapper (DM), tied to a logical volume

The device is rebooted once the wiping activity completes.

Code comparison

When overlapping the flowcharts of AcidRain and AcidPour, as can be seen in the image below, a striking resemblance in the heuristics of both wipers can be observed. Heuristic similarity does not necessarily result in re-use and/or overlap of code, and even if that were to be the case, the overlap could come from benign libraries.

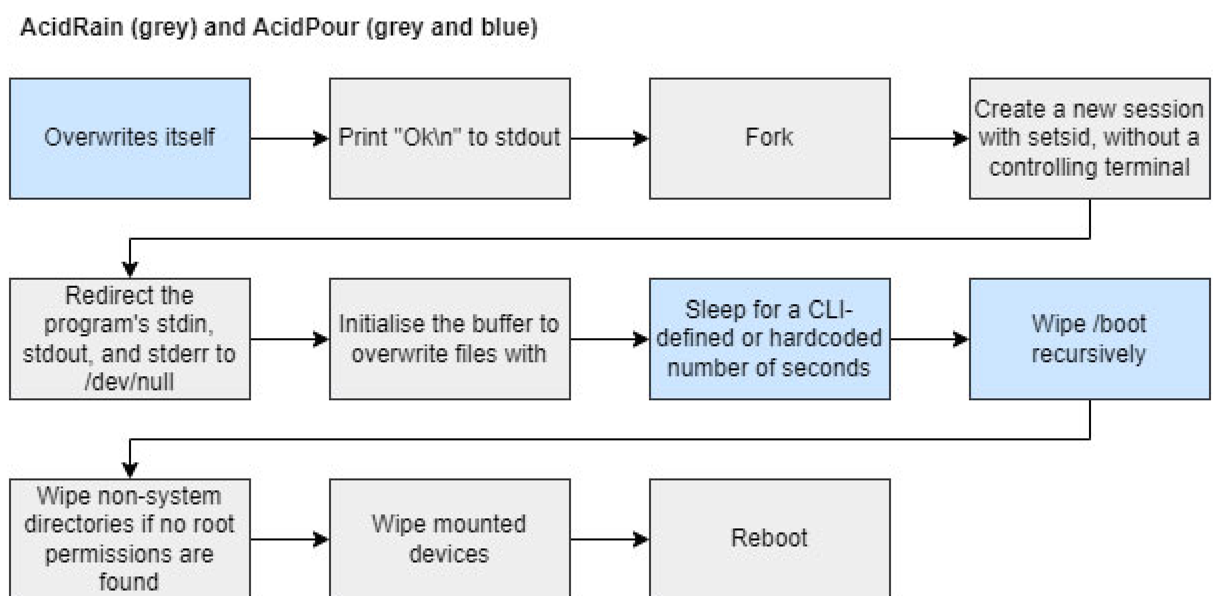

Figure 9 - The overlap between the two wipers in a flow chart

Library code

The two wipers both use code which exists in uClibc. The uClibc-ng (next generation) library is built on uClibc, and thus overlaps. The goal here is not to dive into the difference between these two libraries, but rather that the Acid-wipers used either of these libraries, which explains some of the used code constructs, such as the forking, creation of a new session, and the redirection of the program's standard in- and outputs. These are all part of the library's code within the daemon function. The image below shows the uClibc source code, and the decompiled code from the AcidRain and AcidWiper respectively in a side-by-side comparison. Note that due to compiler optimization, the daemon function's content is inlined within the caller, which is the main function within each Acid-wiper.

```
int daemon(int nochdir, int noclose)
{
    int fd;

    if (fork_parent() == -1)
        return -1;

    if (setsid() == -1)
        return -1;

    if (!nochdir)
        chdir("/");

    if (!noclose)
    {
        struct STAT st;

        if ((fd = open_not_cancel_2(_PATH_DEVNULL, O_RDWR)) != -1
            && (__builtin_expect (FSTAT (fd, &st), 0) == 0))
        {
            if (__builtin_expect (S_ISCHR (st.st_mode), 1) != 0) {
                dup2(fd, STDIN_FILENO);
                dup2(fd, STDOUT_FILENO);
                dup2(fd, STDERR_FILENO);
                if (fd > 2)
                    close(fd);
            } else {
                /* We must set an errno value since no
                   function call actually failed.  */
                close_not_cancel_no_status (fd);
                __set_errno (ENODEV);
                return -1;
            }
        } else {
            close_not_cancel_no_status (fd);
            return -1;
        }
    }
    return 0;
}
```

```
Decompile: main - (acid_rain)
12  write(STDOUT,"Look out!\n\n",10);
13  forkPid = fork();
14  if (1 < forkPid + 1U) goto LAB_exit;
15  setsid();
16  fdDevNull = open("/dev/null",O_WRONLY);
17  if (fdDevNull < 0) goto LAB_close;
18  dup2(fdDevNull,stdin);
19  dup2(fdDevNull,stdout);
20  dup2(fdDevNull,stderr);
21  if (2 < fdDevNull) {
22    close(fdDevNull);
23  }
24  initResult = initialiseDescendingBuffer();
```

```
Decompile: main - (acid_pour)
15  overwriteSelf();
16  write(1,"Ok\n",3);
17  forkPid = fork();
18  if (forkPid + 1U < 2) {
19    setsid();
20    fdDevNull = open("/dev/null",1);
21    if (fdDevNull < 0) {
22      close(0);
23      close(1);
24      close(2);
25    }
26    else {
27      dup2(fdDevNull,0);
28      dup2(fdDevNull,1);
29      dup2(fdDevNull,2);
30      if (2 < fdDevNull) {
31        close(fdDevNull);
32      }
33    }
34    initResult = initialiseAscendingData();
```

Figure 10 - Code overlap with uClibc and the Acid wipers

Another example is the sleep function, called in AcidPour's main function, which uses select to monitor non-existing (NULL) file descriptors, and thus wait for a given period of time. Other researchers have noted that this technique is uncommon. And while it is indeed uncommon, the uClibc library uses this exact code on non-BSD, non-POXIS98, and non-realtime builds. The image below shows the uClibc sleep function's source code and relevant decompiled AcidPour code respectively.

Note that the microseconds ("tv_usec") field is not used in the decompiled code, and the value passed to the function is in seconds, rather than microseconds. This could be done with the help of constant folding and propagation, a compiler optimisation technique where the actual value of variables is precalculated, leaving the compiler to only output the optimized code. As such, the program's execution is not slowed down by redundant calculations without impacting the code's behavior.

```c
int usleep (__useconds_t usec)
{
    struct timeval tv;

    tv.tv_sec = (long int) (usec / 1000000);
    tv.tv_usec = (long int) (usec % 1000000);
    return select(0, NULL, NULL, NULL, &tv);
}
```

```
Decompile: sleep -  (acid_pour)
1
2  void sleep(__time_t seconds)
3
4  {
5    timeval timeVal;
6
7    timeVal.tv_usec = 0;
8    timeVal.tv_sec = seconds;
9    select(0,NULL,NULL,NULL,&timeVal);
10   return;
11 }
```

Figure 11 - The uClibc usleep function from its source code, and the sleep function within AcidPour

Aside from the source code overlap, several FunctionID libraries from Mich on GitHub match with functions within the Acid-wipers, as can be seen in the strncpy excerpt below. The GitHub page shows the inclusion of uClibc binaries for a variety of architectures.

```
**************************************************************
* Library Function - Single Match                            *
*  strncpy                                                   *
*                                                            *
* Library: uclibc 0.9.30.1 binaries                          *
**************************************************************
char * __stdcall strncpy(char * __dest, char * __src, si...
   assume ISA_MODE = 0x0
   assume PAIR_INSTRUCTION_FLAG = 0x0
   assume t9 = 0x402620
char *       v0:4       <RETURN>
char *       a0:4       __dest
char *       a1:4       __src
size_t       a2:4       __n
strncpy                               XREF[3]:   wipeFilesRecursively:00
                                                 deleteNonSystemDirector
                                                 00444a30(*)
00402620 2c c2 00 04    sltiu    v0,__n,0x4
   assume t9 = <UNKNOWN>
00402624 14 40 00 1b    bne      v0,zero,LAB_00402694
00402628 24 83 ff ff    _addiu   v1,__dest,-0x1
0040262c 00 06 38 82    srl      a3,__n,0x2
```

```
**************************************************************
* Library Function - Single Match                            *
*  strcpy                                                    *
*                                                            *
* Library: uclibc 0.9.30.1 binaries                          *
**************************************************************
char * __cdecl strcpy(char * __dest, char * __src)
char *      EAX:4          <RETURN>
char *      Stack[0x4]:4   __dest              XREF[2]:   0804a151(R),
                                                          0804a15b(R)
char *      Stack[0x8]:4   __src               XREF[1]:   0804a14d(R)
strcpy                                   XREF[4]:   handleDevice:08048557
                                                    handleDevice:080485d3
                                                    recursivelyWipe:08048
                                                    wipeBlockDevices:0804
0804a148 57             PUSH    EDI
0804a149 56             PUSH    ESI
0804a14a 83 ec 04       SUB     ESP,0x4
0804a14d 8b 74 24 14    MOV     ESI,dword ptr [ESP + __src]
0804a151 8b 7c 24 10    MOV     EDI,dword ptr [ESP + __dest]
```

Figure 12 - FunctionID matches for uClibc code in AcidRain (left) and AcidPour (right)

Wiper related code overlap

Recognizing the library code makes the analysis easier, and provides a link between two files, but said link is not conclusive. The overlap in heuristics, when removing the uClibc aspects, is best shown in a flowchart, as can be seen below.

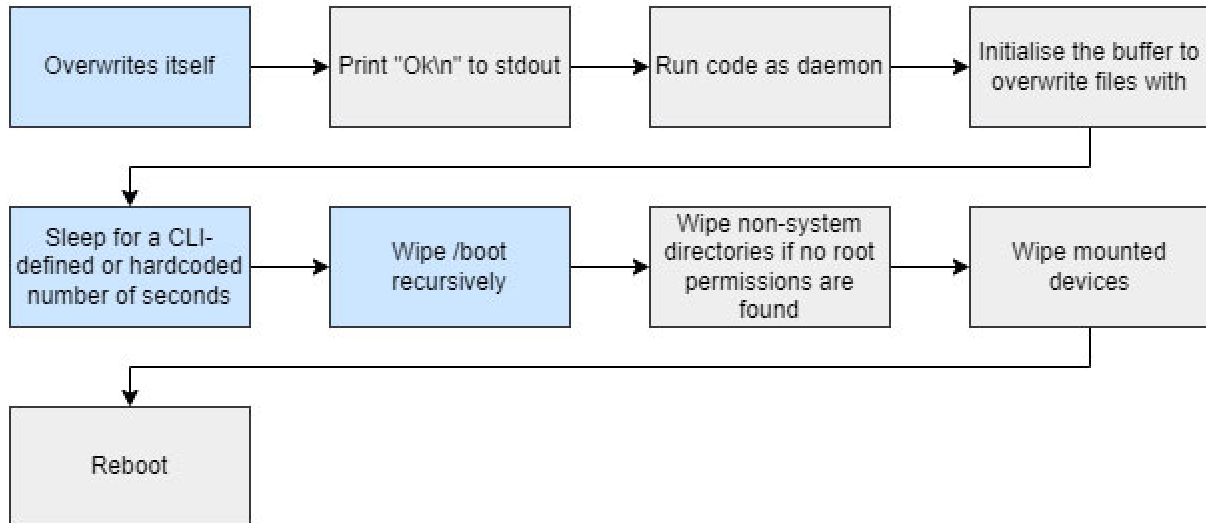

Figure 13 - Overlap between the Acid wipers, taking library code into account

The first aspect which is characterizing the Acid-wipers, is the data that is used to overwrite files with. The AcidRain wiper uses a buffer with descending values, whereas AcidPour uses ascending values. The buffer size in both wipers is 256 kilobytes (0x40000 bytes), the initialization for both can be seen below. Note that the self overwrite function in AcidPour also uses an ascending buffer, as described before.

Decompile: initialiseDescendingBuffer - (acid_rain)

```c
int initialiseDescendingBuffer(void)

{
  int *buffer;
  int value;
  int *end;

  buffer = (int *)malloc(0x40000);
  bufferDescendingValues = buffer;
  if (buffer != NULL) {
    end = buffer + 0x10000;
    if (buffer < end) {
      value = -1;
      do {
        *buffer = value;
        buffer = buffer + 1;
        value = value + -1;
      } while (buffer < end);
    }
    return 0;
  }
  return -1;
}
```

Decompile: initialiseAscendingData - (acid_pour)

```c
int initialiseAscendingData(void)

{
  int *pBuffer;
  int result;
  int i;
  int *pBufferEnd;

                    /* 256 kilobytes */
  bufferAscendingData = malloc(0x40000);
  result = -1;
  if (bufferAscendingData != NULL) {
                    /* 32 kilobytes */
    bufferZeroes = malloc(0x8000);
    pBufferEnd = bufferAscendingData + 0x10000;
    if (bufferAscendingData < pBufferEnd) {
      i = 0;
      pBuffer = bufferAscendingData;
      do {
        *pBuffer = i;
        pBuffer = pBuffer + 1;
        i = i + 1;
      } while (pBuffer < pBufferEnd);
    }
    result = 0;
  }
  return result;
}
```

Figure 14 - Similarities in the creation of the buffer used to wipe files with in AcidRain (left) and AcidPour (right)

The main functions of both wipers also look similar, but due to their brevity and rather simple nature, it is difficult to draw meaningful conclusions from them. A section of the code in both functions is related to the inlining of uClibc's daemon function. Other than that, the printing of a message, the call to several functions to wipe directories, and the call to reboot the device, can easily be recreated without source code access, especially when using the same library code.

However, the wiping methods which are called, do share sufficient logic to state that the author of AcidPour either had access to AcidRain's source code, or that the author simply recreated AcidRain's logic. The input/output control (IOCTL) wiping method (which SentinelOne noted overlapped with the third stage 'dstr' wiper, part of the VPNFilter malware family) employs a similar wiping method but does differ when looked at in more detail. Note that VPNFilter's 'dstr' also uses uClibc.

The OpenWRT project, a Linux operating system for embedded devices, contains code to erase Memory Technology Devices (MTDs), using mtd_info_user and erase_info_user structures. The specific wipe functions within AcidRain, AcidPour, and VPNFilter's 'dstr' all use these structures in a similar manner, as can be seen in the respective side-by-side comparison in the image below.

Figure 15 - The IOCTL wipe functions in (from left to right) AcidRain, AcidPour, and VPNFilter

The used IOCTL calls return these structures in the output argument, meaning that the usage of these structures does not necessarily indicate code overlap with one another, nor with the OpenWRT codebase.

The two Acid wipers have an identical way of wiping code, using the same logic and loops internally. As such, it is likely that the Acid wipers are based on the same codebase, or made by someone with the intention to copy AcidRain when writing AcidPour. The VPNFilter stage uses different logic, indicating that the used source code is likely different, or an older version, compared to AcidRain.

The other wiper related functions show similar overlap, but not all of them are as 'picturesque' as the example taken, making their explanation a tedious process. The above serves as an illustration of reused code within the Acid wipers.

Conclusion

There is no conclusive evidence to state that the Acid wipers are based on the same source code, but given the same dependencies, as well as the similar looking code, we state with medium confidence that AcidPour uses AcidRain's source code as a basis. The modifications allow the wiper to be used in a more versatile manner, as the possible number of systems that it could affect greatly increased.

While the Acid-wipers have been targeting Ukrainian-based entities, we do not attribute it to a specific group. Based on the targeted geographic area, and the critical sectors of publicly reported victims, we attribute these attacks to a pro-Russian actor.

Code overlap comparisons rarely lead to definitive conclusions, given that it is possible to recreate malware based on reversing efforts. The fact that one would have to put in a lot of effort to do so, does not make it impossible, merely unlikely. The (alleged) likeliness can be utilized by an attacker, be it to shroud themselves in smoke and mirrors and throw researchers off their course, or to ensure analysts jump to premature conclusions.

This document and the information contained herein describes computer security research for educational purposes only and the convenience of Trellix customers.

Get the latest

We're no strangers to cybersecurity. But we are a new company.
Stay up to date as we evolve.

Please enter a valid email address.

Zero spam. Unsubscribe at any time.