

# **VisualBoyAdvance & VisualBoyAdvance-M – vulnerabilities in ELF file parser allow for code execution and information disclosure**

TheZZAZZGlitch, 2018

## **Abstract**

VisualBoyAdvance (VBA) is an open-source software emulator for Nintendo's handheld video game consoles, the GameBoy Color and the GameBoy Advance. VBA has been discontinued in 2004 and the original version is no longer maintained. However, many attempts have been made to adopt the abandoned codebase and continue the project. VisualBoyAdvance-M (VBA-M) is a modern counterpart to the original VisualBoyAdvance emulator, which continues to be actively developed to this date.

The software supports loading files in several different formats – this includes the ELF file format. To achieve this, VBA, and consequently VBA-M, both implement a highly simplistic, custom parser for ELF files. Several security vulnerabilities exist in this crude implementation, ranging from low to high severity. Some of them can be exploited to leak information from the host to the emulated environment. Under the right conditions, code execution is also possible, allowing the emulated binary to fully control the host system.

This document describes the found vulnerabilities and provides practical examples of exploitation whenever possible. Note that VBA-M is able to run on several different platforms, but all exploitation examples will assume the code runs under Windows – the dominant operating system amongst the emulator's user base.

## **Short description of the ELF file format**

The Executable and Linkable Format (ELF) is a common file format for executable files. VBA and VBA-M support loading ELF files created for the GameBoy Advance's ARMv4 architecture. A valid ELF file should start with a *file header*, containing basic information about the executable. The file header then provides pointers to *program header table* and *section header table*. Each entry in those tables is used to describe the process of loading the executable – loading data into appropriate memory locations, or providing additional information about the executable, like debug symbols. Full explanation of the ELF file format is out of scope for this document, but those basic concepts are enough to understand the presented vulnerabilities.

## **Wrong memory accesses with invalid offsets in PT\_LOAD segments**

A PT\_LOAD entry in the program header table instructs the loader to insert

specific bytes at a specific memory location in the loaded program's address space – a simple binary copy. VBA handles this case as follows:

```
if(READ32LE(&ph->paddr) >= 0x80000000 &&
    READ32LE(&ph->paddr) <= 0x9fffffff) {
    memcpy(&rom[READ32LE(&ph->paddr) & 0x1fffffff],
        data + READ32LE(&ph->offset),
        READ32LE(&ph->filesz));
    size += READ32LE(&ph->filesz);
}
```

VisualBoyAdvance source code, elf.cpp:2662

- GameBoy Advance (the emulated system) has its ROM memory mapped into addresses 0x08000000 through 0x09FFFFFF. The first *if* statement ensures that the code deals only with ROM addresses.
- *ph->offset* should be an offset within the ELF image from which the section data is loaded.
- The *ph->paddr* member specifies the memory address where the section should be loaded to. However, this address is provided with respect to the emulated program's address space – so the obtained address is converted to an array offset. The target array contains the emulated memory (ROM in this case).
- *ph->filesz* is the section's size.

The offsets and addresses are loaded directly from the ELF image and they are not sanitized in any way, excluding the starting *if* statement. There are two potential vulnerabilities here:

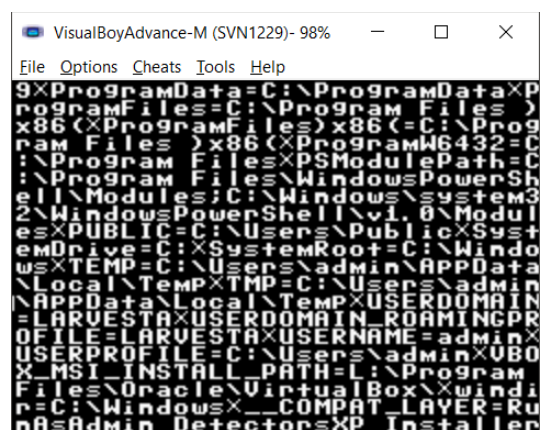
- While the code logic ensures that the loading address is valid, it does not take the section size into account. Loading a section with size exceeding 0x2000000 bytes, would cause data to be copied beyond the destination array (*rom* in this case). This array is allocated on the heap, so the result is a fully controlled heap buffer overflow. This may or may not lead to code execution, depending on the platform the software is running on. Because heap buffer overflows tend to be hard to reliably exploit, this attack vector is not investigated further in this document.
- No checks of any kind are done on the source pointer. By providing a section offset exceeding the size of the loaded ELF file, it's possible to read from unrelated memory areas. This leads to an information disclosure bug, where the emulated image can read arbitrary data from the emulator's process memory.

## **Demonstration of the above vulnerability**

Although the offset can be picked arbitrarily, it is applied to a heap memory location, which is affected by ASLR on most modern platforms. This makes it hard to predict the exact source address. However, the impact of ASLR can be dramatically reduced by creating a very large ELF image (more than 256MB). Such allocation requires a big, contiguous area of memory – the first such area in VBA and VBA-M's process memory starts around address 0x10000000. So within a margin of error caused by ASLR, the image is guaranteed to be loaded around this memory address.

To show this, a simple demonstration has been created: a 512MB ELF file, containing a PT\_LOAD segment in its program header, instructing the loader to read 0x1000 bytes from file offset 0x67000000. This, together with the 0x10000000 base address, should yield the source address of 0x77000000, which fits into the high memory range used by system DLLs on Windows. Adding in the randomness of ASLR, this should result in an emulated application randomly reading data from memory related to system libraries. The ELF file also contains a regular code section with a simple program that prints the leaked data as an ASCII dump.

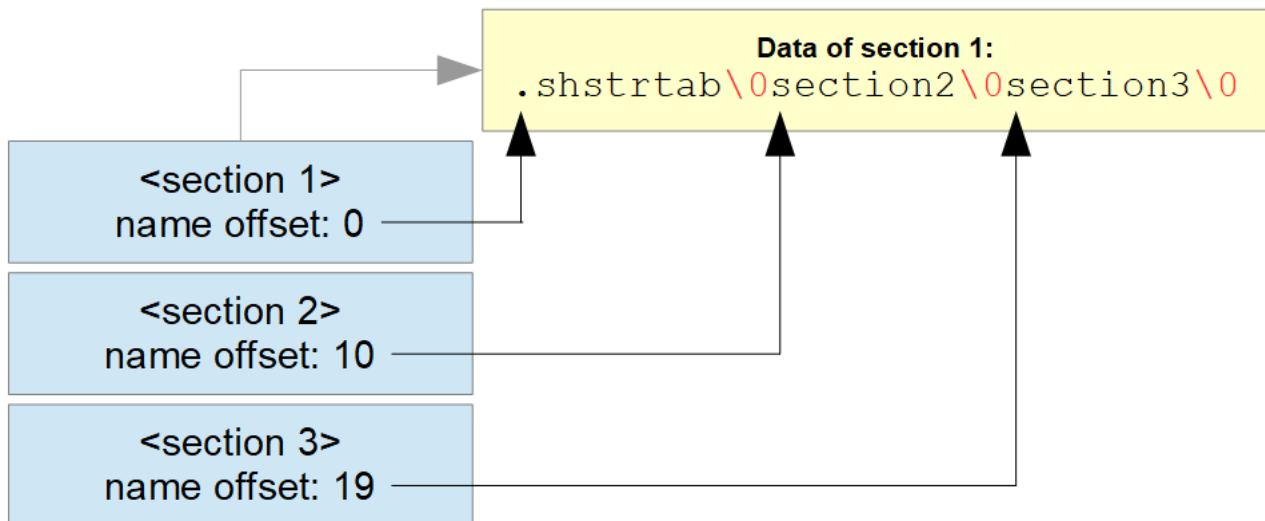
The results were highly promising on both VBA and VBA-M. Only a small percentage (12%) of executions crashed the emulator. Even with the random nature of the exploit, leaked strings often contained operating system versions, environment variables, path to the current directory, and other meaningful information.



The ELF file used in the above demonstration, along with its source code, should be enclosed with this document.

## Wrong memory accesses with invalid offsets to section names

Every section defined in the ELF section table should have a name. All names are stored as a list of null-terminated strings in a separate section, traditionally called `".shstrtab"`. Each entry in the section table contains an offset from which the section name should be read.



The `e_shstrndx` field in the program header should determine the index of the section header table entry that contains the section names. Sections are indexed starting from 1. A value of 0 indicates that the loaded image has no sections (and thus, no section name table). VBA handles this case as follows:

```
char *stringTable = NULL;
if(READ16LE(&eh->e_shstrndx) != 0) {
    stringTable = (char *)elfReadSection(data,
                                         sh[READ16LE(&eh->e_shstrndx)]);
}
```

VisualBoyAdvance source code, elf.cpp:2670

If `e_shstrndx` is set to 0, the string table pointer remains set as NULL. This is normally irrelevant – if the image has no sections, no operation will ever be performed on that pointer. However, it is possible to create a malformed file that both contains a nonzero number of sections, and sets the name table index to 0. This would cause a null pointer dereference the first time a section name is loaded. Additionally, by providing specially crafted name offsets in section header entries, it's possible to point a section name anywhere in memory and create arbitrary memory reads.

It's unfortunately impossible to access section names from within the emulated code. Additionally, there are no emulator features that directly deal with section names. Therefore, this attack vector doesn't seem particularly useful. It is however included here, as its potential impact may change as new software versions are released.

## **Multiple buffer overflow vulnerabilities while handling ELF file symbols**

ELF files can also contain debugging symbols. The exact method of including symbol information in ELF images depends on the compiler used to link the program. VBA and VBA-M both support loading symbols encoded in ".symtab" sections, which is a common method of introducing debug information to ELF binaries.

The most vital piece of information about any symbol is its name. The ELF file loader module in VBA has a function designed to read the name of any symbol, based on its address:

```
char *elfGetAddressSymbol(u32 addr)
{
    static char buffer[256];
    (...)
    if(addr == s->value) {
        if(s->name)
            strcpy(buffer, s->name);
        else
            strcpy(buffer, "");
        return buffer;
    }
}
```

VisualBoyAdvance source code, elf.cpp:285

No symbol name limit is enforced at any time, yet the name is loaded into a static 256-byte buffer, resulting in a typical buffer overflow scenario. This overflow would be capable of overwriting a whole slew of global variables, most likely allowing for code execution. However, to exploit this potential vulnerability, this code has to be called from somewhere else. It turns out this function is used in two places – the in-built disassembler, to display symbols in the disassembly, and in the in-built gdb debugging server, presumably for the same purpose.

```
Search "elfGetAddressSymbol(" (10 hits in 3 files)
W:\code\vba\visualboyadvance-m\src\gba\armdis.cpp (3 hits)
Line 599:      const char* s = elfGetAddressSymbol(value);
Line 609:      const char* s = elfGetAddressSymbol(value);
Line 696:      const char* s = elfGetAddressSymbol(offset + 4 + add);
W:\code\vba\visualboyadvance-m\src\gba\elf.cpp (2 hits)
W:\code\vba\visualboyadvance-m\src\sdl\debugger.cpp (5 hits)
Line 901:      elfGetAddressSymbol(debuggerBreakpointList[i].address));
Line 1363:     size_t l = strlen(elfGetAddressSymbol(pc + 4 * i));
Line 1371:     fprintf(f, format, addr, elfGetAddressSymbol(addr), buffer);
Line 1382:     size_t l = strlen(elfGetAddressSymbol(pc + 2 * i));
Line 1391:     fprintf(f, format, addr, elfGetAddressSymbol(addr), buffer);
```

The disassembler seemed like an easier target, so this code path was chosen for further analysis. It turned out, exploitation of this vulnerability was not even necessary. The disassembler's GUI routine happened to have a different buffer overflow vulnerability, even featuring a source code comment claiming that "the buffer is definitely large enough, no need to worry".

```

// what an unsafe calling convention
// examination of disArm shows that max len is 69 chars
// (e.g. 0x081cb6db), and I assume disThumb is shorter
char buf[80];
(dest points to buf at this point...)
const char* s = elfGetAddressSymbol(value);
if (*s) {
    *dest++ = ' ';
    dest = addStr(dest, s);
}

```

VisualBoyAdvance-M source code: viewers.cpp:146, armdis.cpp:599

This one is a stack-based buffer overflow. On classic VBA it is trivial to exploit, since none of the modern security measures are in place. On VBA-M, the stack cookie mechanism prevents exploitation of this particular vulnerability. Additionally, in VBA-M's code, the *strcpy* and *sprintf* calls in *elfGetAddressSymbol* were replaced by their "safe" counterparts, *strncpy* and *snprintf*, so the original vulnerability is not present too.

An example, proof of concept exploit should be enclosed to this document. It works by overwriting the return address with 0x20202020, and uses the trick described earlier to allocate the ELF image around address 0x10000000. The 512MB file contains a NOP sled, with a calc.exe shellcode stub at the end.

In order to trigger the vulnerability, the user has to: Load a malicious ELF file, open the in-built disassembler by selecting *Tools » Disassemble*, then click "Goto R15".

## **Vendor status**

The mainstream version of VBA has been officially discontinued for more than 10 years, making any form of vendor-coordinated disclosure impossible. VBA-M is still being actively worked on, so their team has been informed about the found issues before publishing the document.

- 13.06.2018 – the vulnerabilities have been reported
- 26.06.2018 – a fix has been created and scheduled for release
- 01.07.2018 – releasing VBA-M version 2.1.0, which contains the aforementioned fix
- 04.07.2018 – publishing this document

## **Disclaimer**

This document, along with all information it contains, is provided "as is", without any warranty. The author is not responsible for the misuse of the information provided in this document. Permission is granted to redistribute this document, as long as its content and copyright notices remain intact.