

Binary Hacking: X86 VM Sandbox Escape

Tizian Seehaus Athanasios Tsarapatsanis Murtaza Haidari
Beginner's Practical, B.Sc. Computer Science
Heidelberg University, Germany

Abstract—Our objective was to design an exploit within a memory management unit (MMU), which necessitated the development of our own emulated central processing unit (CPU).

I. CPU EMULATOR - ARCHITECTURE

A. Overview

In developing the CPU emulator (CPUE), we adhered to the principles and methodologies characteristic of Intel architecture. This approach led us to incorporate the following modules (Fig.[1]):

- 1) **Embedded Arithmetic Logic Unit (ALU):**
Responsible for executing arithmetic and logical operations within the CPUE.
- 2) **Memory Management Unit (MMU):**
Manages memory access and translation between virtual and physical addresses.
- 3) **Translation Lookaside Buffer (TLB):**
Caches recent address translations to expedite memory access.
- 4) **Memory-Mapped Input/Output (MMIO):**
Facilitates communication between the CPU and peripheral devices via designated memory addresses.
- 5) **Programmable Interrupt Controller (PIC) & Interrupt Control Unit (ICU):**
Handles interrupt signals to ensure proper CPU response to asynchronous events.
- 6) **Universal-Asynchronous-Receiver-Transmitter (UART) Controller:**
Manages serial communication by converting parallel data from the CPU into serial form and vice versa.
- 7) **V100-Terminal:**
Emulates a terminal interface for user interaction with the CPUE.

B. Modules

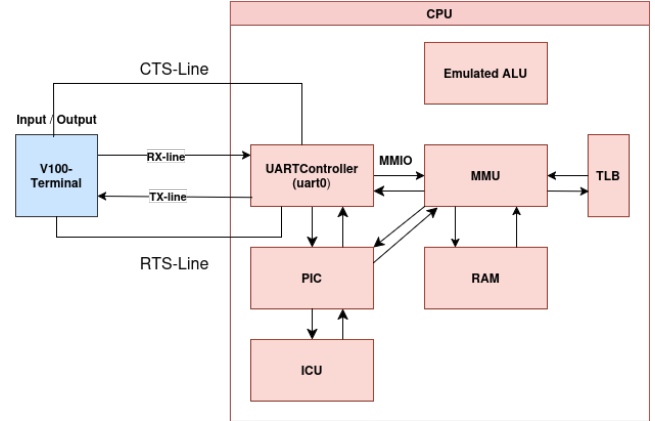


Fig. 1. Overview of CPU modules

1) **Embedded ALU:** This component represents the CPUE's internal logic capable of performing arithmetic and logical operations. Given the project's scope, implementing individual logic gates was deemed unnecessary. Instead, we implemented the official Intel instruction set architecture (ISA) [1] to define the ALU's functionality.

2) **MMU:** Our MMU implementation encompasses comprehensive memory management functionalities, excluding protected keys. It effectively handles page faults by delegating them to interrupt handlers, as detailed in the PIC section. The MMU incorporates an interrupt descriptor table and utilizes gate descriptors within the interrupt handler, supporting call, interrupt, and trap gates. It employs a four-level paging mechanism, aligning with Intel's architecture (Fig.[2]), and supports segmentation through logical addresses. Notably, the following code example shows where our bug is hidden (List.[1]).

Listing 1. Exploit inside of the MMU

```
template<typename T = u8>
T* paddr_ptr(PhysicalAddress const& paddr) {
    CPUE_ASSERT(m_physmem != NULL, "m_physmem==NULL");
    return (T*)(m_physmem + paddr.addr);
}
```

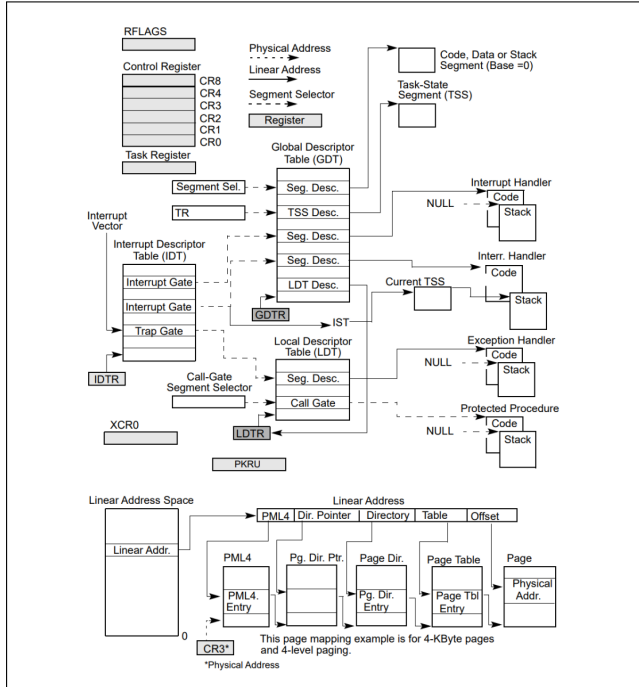


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode and 4-Level Paging

Fig. 2. General paging structure

3) **TLB:** The TLB is designed following Intel's standards to enhance MMU performance by caching recent address translations, thereby reducing memory access latency.

4) *MMIO*: This module maps registers inside of the PIC and the UART-controller to the MMU, facilitating communication between hardware components and the CPU. This ensures for a data exchange between the MMU and the PIC, as well as between the MMU and the UART-controller. (Fig. [3])

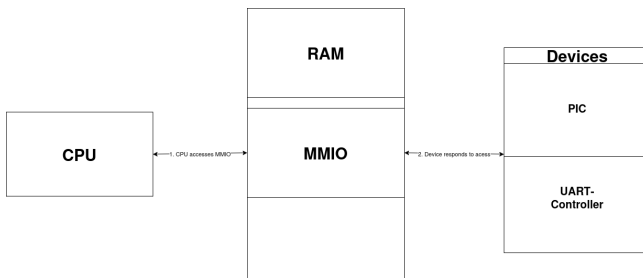


Fig. 3. Depiction of MMIO

5) *PIC & ICU*: The design of our PIC, was inspired by the 8259A PIC. Our system features extensive interrupt handling capabilities, allowing for nested interrupts that are processed according to their priority levels. When an interrupt occurs during the handling of another, the system manages them based on predefined priorities. Additionally, we support hardware interrupt logic, which necessitates management by the operating system's kernel [2].

6) *UART-Controller*: The design pattern was inspired by the TL16C750 model. It facilitates serial communication through Receive/Transmit lines, along with clear-to-send and ready-to-send lines. Regarding the original design ours only supports the FIFO-mode [3].

7) *V100-Terminal*: The V100-Terminal emulates a simple terminal interface, enabling user interaction by providing input and output functionalities. It also provides an internal FIFO-buffer which caches in characters that are not ready to be received by the UART-controller.

C. Executing the CPU-Emulator

When operating the CPUE with a mini-kernel, we successfully executed various programs. This includes running tiny BASIC programs, such as a simple bubble-sort algorithm, as well as applications like 'cowsay' and others that require minimal system calls (Fig. [4] & [5]).

```

1 $ ./cpw -kernelcustom --kernel-ldm ../minik/build/minik-ldm ../minik/build/tinybasic/tiny-basic2/bas2c2b2devout.elf
You are about to call the entry point of your binary-elf.
How many arguments do you want to pass to it? 0
Let's go :)

2
3
4
5
6
7
8
9
10 print "Unsorted:"
11 for i = 0 to 9 : @() = rnd(100) : print @() : next i
12 d = 1
13 for i = 1 to 9
14   if @() < @() - 1 then gosub 90
15 next i
16 if d = 0 then goto 30
17 goto 100
18 s = @() : @() = @() - 1 : @() - 1 = s : d = 0 : return
19 print
20 print "Sorted:"
21 for i = 0 to 9 : print @() : next i > > > > > > > > >
22
23 run
24
25 Unsorted:
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
99
```

Fig. 4. Executing bubble-sort on tiny-basic

```
# ./cspaw -kernelcustom --kernel-lag ../bin/kbuild/atn-k-lag ../bin/kports/cowsay/ccowsay/cowsay %0 >/dev/null
You are about to call the entry point of your binary-elf.
How many arguments do you want to pass to it? 1
Please input argument 1: Neo would be proud - I'm out of the Matrix!
Let's go :)

Neo would be proud - I'm out of the Matrix! >
-----
      \  (oo)\_____/
       (__)\       )\/\
        ||----w |
        ||     ||
```

Fig. 5. Executing cowsay

II. EXPLOITING THE EMULATOR

A. The idea behind the exploit

Our exploit draws inspiration from the return-to-libc attack technique. Specifically, we aim to manipulate the `exit_func_list` entries to include a call to `system("/bin/sh")`. This manipulation is facilitated by the given vulnerability within our Memory Management Unit (MMU) implementation, which fails to enforce proper bounds checking on physical addresses. Consequently, this oversight permits an overflow of the emulated RAM, allowing access to the standard C library (libc) within the process's memory space. Notably, our approach does not involve disabling existing security mechanisms such as stack canaries, Address Space Layout Randomization (ASLR), etc. (Fig. [6])

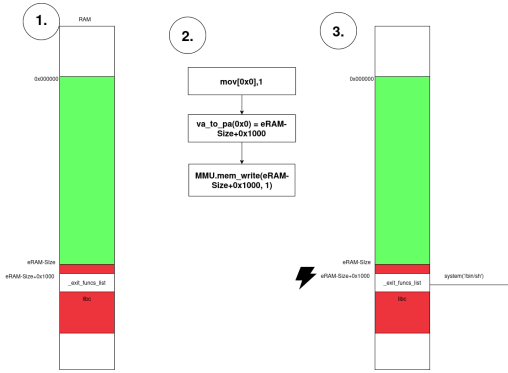


Fig. 6. Overview of the stack

B. How to run the exploit

1) *The Decryption of mangled pointer:* In order to execute the exploit, we must first locate the relative offset of the libc-library to our emulated RAM. This step is crucial because ASLR is randomizing the address each time we run the program. To achieve this we randomly jump out of the emulated RAM's bounds and try calculate the relative offset. When inside of libc we can now read the entries of the global offset table (GOT) and bypass ASLR.

After finding the libc base we can now calculate the location of `__exit_function_list` and thus search for the entry `_dl_fini`. We can detect `_dl_fini` as it's values are `flavor=ef_cxa`, `arg=0`, `dso_handle=0`. By finding `_dl_fini`, which is the secret for mangling the fn-pointer, we are now able to decrypt the fn-pointer. Decrypting the fn-pointer is possible by reversing the encryption-steps ([Encryption of fn-pointer](#)). So in our case we have to do a rotation by `0x11` of the fn-pointer and then XOR it with `dl_fini_addr`.

2) *Overwriting exit_function:* Now we can overwrite the `exit_function.fn` member of the `exit_function` struct entry corresponding to `_dl_fini` with our own correctly mangled system function pointer. As an argument we set `exit_function.arg` to the address of the string `"/bin/sh"`, which is conveniently already present inside the libc data segment. This results in an overall effect wherein the libc exit handler routines will handle our faked entry and call `system("/bin/sh")`, giving us remote code execution on the hypervisor host machine, and with this completing our x86 vm sandbox escape.



Fig. 7. Overview of exit_func_list

```

python3 exploit.py build/exploit
[*] Opening connection to localhost on port 1024: Done
Press enter to send the END sequence...
[*] Switching to interactive mode

[*] Kernel initialized
[*] Finding libc.....
[*] Found an elf page at offset 0x139000. Now searching for libc elf.....
[*] Found libc elf at offset 0x218000!
[*] Got ld_base @ 0x793f1b637000
[*] Got libc_base @ 0x793f1b218000
[*] Got dl_fini_addr @ 0x793f1b63b680
[*] Got system_addr @ 0x793f1b264490
[*] Got /bin/sh string addr @ 0x793f1b3ae031
[*] Searching for _dl_fini entry in initial exit_function_list.....
[*] Found _dl_fini entry at index 9
[*] Got pointer_guard: 0xff5488944d125f8
[*] PTR_MANGLED(system): 0xe356bfeec2d1fea9
[*] Overwriting _dl_fini entry with system('/bin/sh')..... - Done!
[*] Exiting emulator and trigger the exploit... enjoy :)
whoami
chall
cat flag
CTF{This_hypervisor_aln't_hyper_enough_to_stop_me!}

```

Fig. 8. Running the Exploit

REFERENCES

- [1] *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, Intel Corporation, December 2024, available at: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [2] *Intel® 8259A Programmable interrupt controller (8259A/8259A-2)*, Intel Corporation, available at: <https://pdos.csail.mit.edu/6.828/2010/readings/hardware/8259A.pdf>.
- [3] *PC16550D Universal Asynchronous Receiver/Transmitter With FIFOs*, Texas Instruments Incorporated, available at: <https://media.digikey.com/pdf/Data%20Sheets/Texas%20Instruments%20PDFs/PC16550D.pdf>.