Chair of IT-Security
Department of Informatics
Technical University of Munich

Dynamic Loader Oriented Programming on Linux

Julian Kirsch, Bruno Bierbaumer, Thomas Kittel, and Claudia Eckert

Reversing and Offensive-oriented Trends Symposium (ROOTS) 2017

Research Question

Are current exploit mitigations capable of detecting and preventing abuse of array out-of-bound write vulnerabilities?

More specifically, can an (artificial) C program (cf. Figure 1) be attacked to gain arbitrary code execution even if common exploit mitigations (cf. Figure 2) are enabled?

Core Ideas

1. If user controlled buffers get allocated next to memory owned by the C runtime environment, it is possible to pre-calculate fixed distances from the beginning to the array to these control structures.
2. Given the ability to corrupt memory used by the C runtime, it is possible to find data structures that can be overwritten with constant values resulting in reliable arbitrary code execution.

Approach

1. Measure distances of newly allocated memory to data structures used by `libc.so.6` and `ld.so`.
2. After identifying allocation strategies that return memory at a fixed distance to `libc.so.6` and `ld.so` (reachable pointers), find writeable data structures within these libraries that are dispatched during program shutdown (defilable pointers) using a combination of taint analysis and program slicing.
3. Manually examine reachable and defilable pointers for instruction slices allowing to bypass Address Space Layout Randomization (ASLR).

Results

1. When ASLR is turned on, the `mmap` system call randomizes the absolute pointer values returned, but not necessarily the relative distances. Figure 3 summarizes our findings for Arch Linux running a 4.12. kernel: For example, memory dynamically allocated by `malloc` with a large size argument (`0x200000`) resides at a constant distance to the writable data region of `ld.so`.
2. Due to the unique structure of how `ld.so` stores information related to destructor handling in writable memory (even if `relro` is active) it is possible to bypass ASLR (and all other mitigations) using only constants to overwrite members of `struct link_map` when exploiting an array out-of-bounds write vulnerability.

Code

`https://github.com/kirschju/wiedergaenger`

```c
1  int main(int argc, char **argv) {
2      /* Exemplary initialization */
3      uint8_t *array = malloc(0x200000);
4      size_t idx = 0, val = 0;
5
6      while (scanf("%zu %zu", &idx, &val) == 2) {
7          array[idx] = val;
8      }
9
10     return 0;
11 }
```

Figure 1: Artificial C program simulating an out-of-bounds write vulnerability in line 7

```
$ gcc vuln.c -Wl,-z,noexecstack,relro,now \
             -pie -fPIC -stack-protector=all \
             -D_FORTIFY_SOURCE=2 \
             -out vuln
```

Figure 2: Compiler invocation to turn on common exploit mitigations

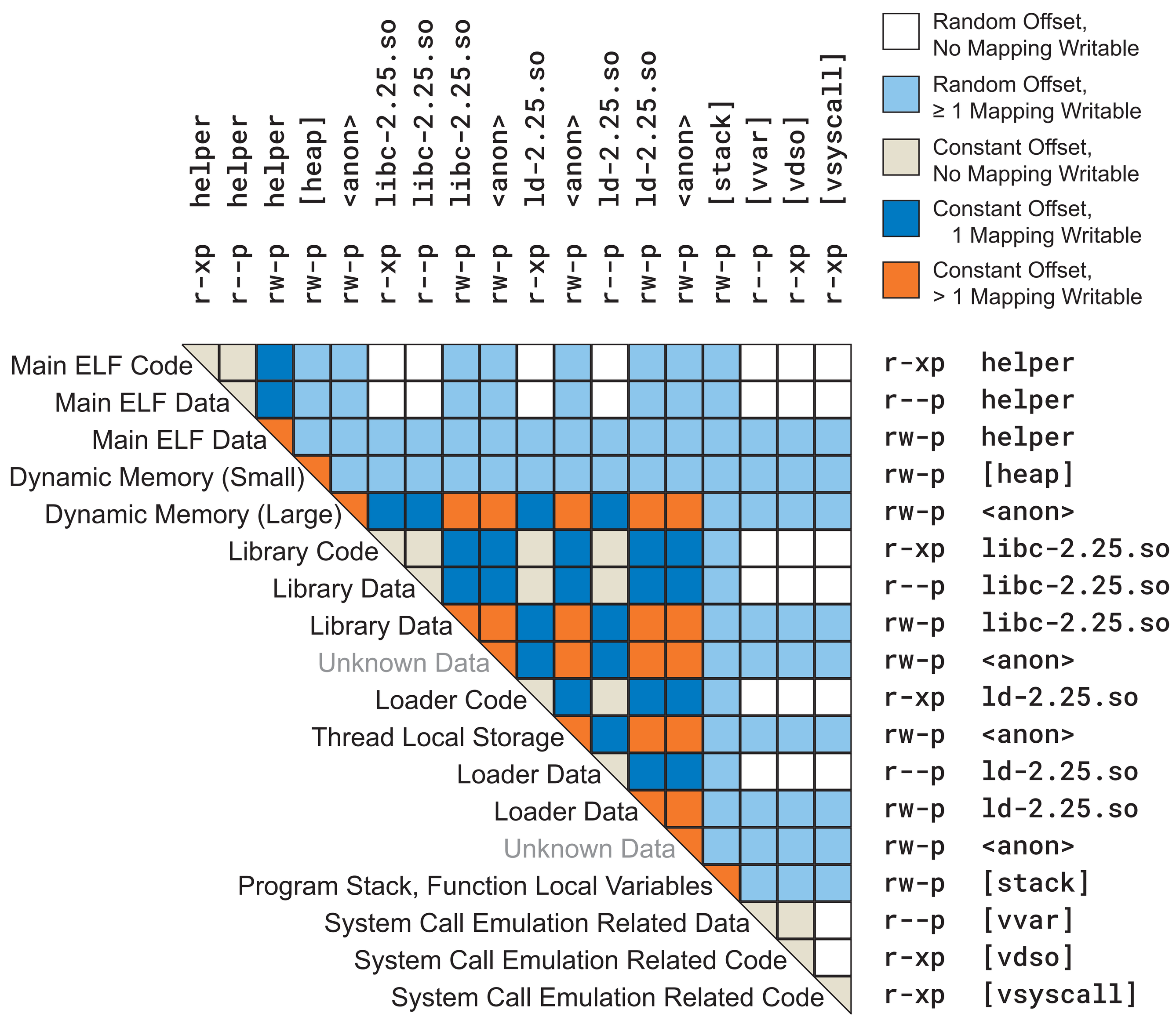

Figure 3: Color matrix showing memory regions sharing constant (□/■/■) or random (□/■) distances with each other for applications running on Arch Linux.

```c
/* Debian 10 kernel 4.12.6-1 (glibc 2.24-17) */
int main(int argc, char **argv)
{
    uint8_t *ptr;
    ptr = malloc(0x200000);

    /* Distance of the malloced pointer to struct link_map used by ld.so */
    size_t base = 0x7c3160;

    /* Set l->l_addr to offset of _r_debug in ld.so to win-gadget in libc.so */
    *(uint64_t *)&ptr[base] = 0xffffffffffb1480f;

    /* Set l->l_info[DT_FINI] pointer to a pointer to _r_debug */
    ptr[base +  0xa8] = 0xb8;

    /* Set l->l_info[DT_FINI_ARRAYSZ] pointer to a value < 8 */
    ptr[base + 0x120] = 0xc0;

    return 0;
}
```

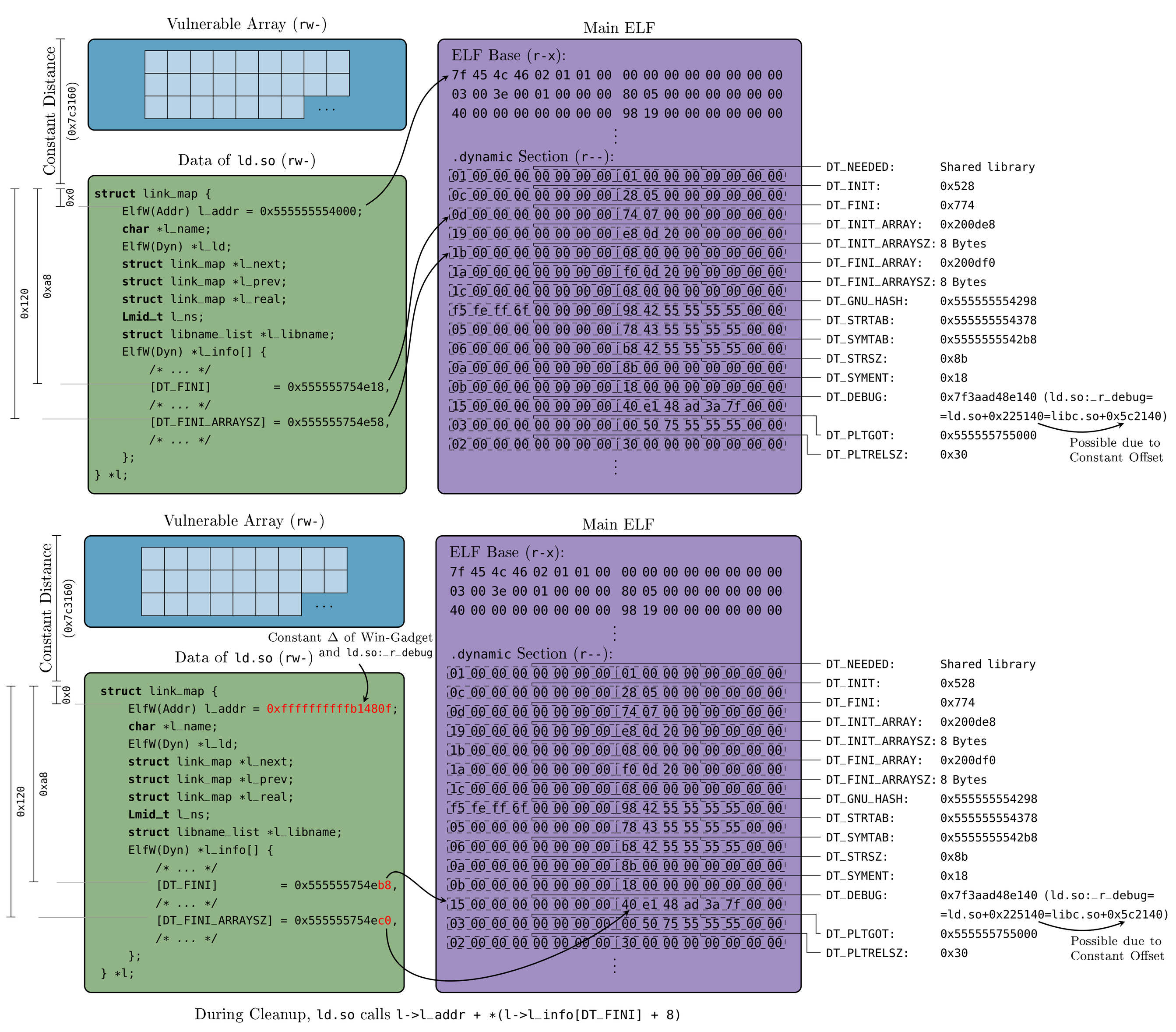

Figure 4: An example of a C program that ends up executing `execve("/bin/sh", argv, envp)` by abusing the fact that the pointer returned by `malloc` has a fixed distance of `0x7c3160` bytes to `struct link_map`, a writable data structure used by `ld.so`. The diagrams at the right side visualize the meaning of the constants used during the corruption. Note that all values can be pre-calculated, regardless of ASLR.

V. i. S. d. P.:
Lehrstuhl für Sicherheit in der Informatik
Fakultät für Informatik
Boltzmannstraße 3
85748 Garching