LibCare

Welcome to the LibCare project documentation. Our aim is to be able to patch any of your executables or libraries at the run, so you don't have to restart your servers whenever a new wild CVE appears.

See how it helps to patch famous GHOST vulnerability resulting in a buffer overflow in the domain name resolution subroutines of `glibc`, skip to Contents or straight to the installation guide.

Sample `samples/server`

For instance, your backend developer made a typo during server development. This typo introduced a stack overflow vulnerability exploitable from the client side. Common automatic checks were disabled for the sake of performance and now your server is vulnerable to anyone who can find the vulnerability.

The sample code is in `samples/server/server.c` where function `handle_connection` supplies wrong buffer size to the `recv(2)` at line 24:

```
void handle_connection(int sock)
{
        char buf[16];

        (void) recv(sock, buf, 128, 0); // bug is here
        fprintf(stdout, "Got %s\n", buf);
        close(sock);
}
```

1. Build the original server and run it:

```
$ cd samples/server
$ make install DESTDIR=vuln
cc -o server server.c -fno-stack-protector -fomit-frame-pointer
$ ./vuln/server
```

2. Now let's install dependencies and build utils. Refer to installation for more details on the installation procedure and supported OSes.

   For RHEL-based distros do:

```
$ sudo yum install -y binutils elfutils elfutils-libelf-devel nc libunwind-devel
...
$ make -C ../../src
...
```

   For Debian-based distros do:

```
$ sudo apt-get install -y binutils elfutils libelf-dev netcat-openbsd libunwind-dev
...
$ make -C ../../src
...
```

3. Try to connect to the server using freshly installed netcat:

```
$ echo "Hi!" | nc localhost 3345
Hi!
```

   The server should print on its console:

```
$ ./vuln/server
Got Hi!
```

4. Now exploit the server via the `hack.sh` script. The script analyzes binary and builds a string that causes server's buffer to overflow. The string rewrites return address stored on the stack with the address of `you_hacked_me` function, which prints "You hacked me!" as a server.

   Open another console and run `./hack.sh` there:

```
$ ./hack.sh
```

   Server console should print:

```
Got 0123456789ABCDEF01234567@
You hacked me!
```

   This sample emulates a packaged binary network server vulnerable to return-to-libc attack.

5. Now build the patch for this code via kpmake:

```
$ ../../src/kpmake --clean server.patch
...
patch for $HOME/libcare/samples/server/kpmake/server is in ...
```

   Please note that this overwrites `./server` binary file with a patch-containing file, storing the original vulnerable server into `./kpmake/server`.

6. Examine `patchroot` directory and find patches there:

```
$ ls patchroot
2d0e03e41bd82ec8b840a973077932cb2856a5ec.kpatch
```

7. Apply patch to the running application via kpatch_user:

```
$ ../../src/kpatch_user -v patch -p $(pidof server) patchroot
...
1 patch hunk(s) have been successfully applied to PID '31209'
```

8. And check the hack again, `You hacked me!` string should go away:

```
(console2) $ ./hack.sh
(console1) $ # with running ./vuln/server
Got 0123456789ABCDEF@
```

Congratulations on going through the sample! Go on and learn how the magic of kpmake script works, read how the patch is built under the hood and how it is applied by the kpatch_user. Or even jump to our hacking guide!

RHEL7 `glibc` sample

Most of the binaries in the system are coming from distribution packages so building patches for them is different from the above. Here is how to do it.

This example builds `glibc` patch for an old fashioned CVE-2015-0235 GHOST vulnerability for RHEL7. The build is done using scripts/pkgbuild and package files are stored in `packages/rhel7/glibc/glibc-2.17-55.el7`.

Preparing environment

First, we need the exact versions of tools and libs. Let's build a Docker image and a container for it:

```
$ docker build docker/kernelcare/centos7/gcc-4.8.2-16.el7 \
        -t kernelcare/centos7:gcc-4.8.2-16.el7
...
$ docker run -v $PWD:/libcare --cap-add SYS_PTRACE -it \
        kernelcare/centos7:gcc-4.8.2-16.el7 /bin/bash
[root@... /]#
```

Now, from inside the container let's install vulnerable version of glibc:

```
[root@... /]# yum downgrade -y --enablerepo=C7.0.1406-base \
        glibc-2.17-55.el7 glibc-devel-2.17-55.el7 \
        glibc-headers-2.17-55.el7 glibc-common-2.17-55.el7
...
```

Build the `libcare` tools:

```
[root@... /]# make -C /libcare/src clean all && make -C /libcare/execve
...
```

Now build and run the sample GHOST app that runs 16 threads to constantly check whether the `glibc` is vulnerable to GHOST and prints a dot every time it detects a buffer overflow in the `gethostbyname_r` function. The downgraded `glibc` is vulnerable:

```
[root@... /]# cd /libcare/samples/ghost
[root@... ghost]# make
...
[root@... ghost]# ./GHOST
............^C
```

Press Ctrl+C to get your console back and let's start building the patch for `glibc`.

Building and applying the patch

The build is done in two stages.

First, the original package build is repeated with all the intermediate assembly files stored and saved for later. This greatly helps to speed up builds against the same base code. Run the following from inside our docker container to pre-build `glibc` package:

```
[root@... /]# cd /libcare/
[root@... /libcare]# ./scripts/pkgbuild -p packages/rhel7/glibc/glibc-2.17-55.el7
...
```

This should download the package, do a regular RPM build with `kpatch_cc` wrapper substituted for GCC and store the pre-built data into the archive under `/kcdata` directory:

```
[root@... /libcare]# ls /kcdata
build.orig-glibc-2.17-55.el7.x86_64.rpm.tgz  glibc-2.17-55.el7.src.rpm
```

Now let's build the patch, the output will be verbose since it contains tests run by the `kp_patch_test` defined in `packages/rhel7/glibc/glibc-2.17-55.el7/info`:

```
[root@... /libcare]# ./scripts/pkgbuild packages/rhel7/glibc/glibc-2.17-55.el7
...
[root@... /libcare]# ls /kcdata/kpatch*
/kcdata/kpatch-glibc-2.17-55.el7.x86_64.tgz
```

Unwrap patches and run the GHOST sample:

```
[root@... /libcare]# cd /kcdata
[root@... /kcdata]# tar xf kpatch*
[root@... /kcdata]# /libcare/samples/ghost/GHOST 2>/dev/null &
[root@... /kcdata]# patient_pid=$!
```

And, finally, patch it. All the threads of the sample must stop when the GHOST vulnerability is patched:

```
[root@... /kcdata]# /libcare/src/kpatch_user -v patch -p $patient_pid \
                root/kpatch-glibc-2.17-55.el7.x86_64
...
1 patch hunk(s) have been successfully applied to PID '...'
(Press Enter again)
[1]+  Done                    /libcare/samples/ghost/GHOST 2> /dev/null
```

You can patch any running application this way:

```
[root@... /kcdata]# sleep 100 &
[root@... /kcdata]# patient_pid=$!
[root@... /kcdata]# /libcare/src/kpatch_user -v patch -p $patient_pid \
                root/kpatch-glibc-2.17-55.el7.x86_64
...
1 patch hunk(s) have been successfully applied to PID '...'
```

Congratulations on finishing this rather confusing sample!

Contents

Installation and dependencies

All the Linux-distros with available `libunwind`, `elfutils` and `binutils` packages are supported.

However, the `libcare` is only tested on Ubuntu from 12.04 to 16.04 and on CentOS from 6.8 to 7.3.

Dependencies

To install the dependencies on RHEL/CentOS do the following:

```
$ sudo yum install -y binutils elfutils elfutils-libelf-devel nc libunwind-devel
```

To install the dependencies on Debian/Ubuntu do the following:

```
$ sudo apt-get install -y binutils elfutils libelf-dev netcat-openbsd libunwind-dev
```

Building `libcare`

To build `libcare` emit at project's root dir:

```
$ make -C src
...
```

This should build all the utilities required to produce a patch out of some project's source code.

It is highly recommended to run the tests as well, enabling Doctor `kpatch_user` to attach `ptrace`cles to any of the processes first:

```
$ sudo setcap cap_sys_ptrace+ep ./src/kpatch_user
$ make -C tests && echo OK
...
OK
```

Now all the required tools are built and we can build some patches. Skip to sample for that.

Overview

First, we prepare project patch by examining the differences in assembler files generated during the original and the patched source code build. Finally, users invoke the `kpatch_user` that applies the patches. This is a lot like loading a shared object (library) into other process memory and then changing original code to unconditionally jump to the new version of the code.

1. Patch preparation
2. Project patch building
3. Patch application

Patch preparation

Binary patches are built from augmented assembly files. Augmented files are made via `kpatch_gensrc` which notes the difference in assembly files produced from the original and the patched source code.

This is done in two steps, both are described detailed in Manual Patch Creation.

Building originals

First, the original code is built as is either by invoking `make` directly or by the packaging system. The build is done with compiler substituted to `kpatch_cc` wrapper. Wrapper's behaviour is configured via environment variables.

When `kpatch_cc` is invoked with `KPATCH_STAGE=original` it simply builds the project while keeping intermediate assembly files under the name `.kpatch_${filename}.original.s` invoking the real compiler twice: first with the `-S` flag to produce the assembly files from the original code and then with the `-c` flag to produce object files out of these intermediate assembly files.

Project binaries built during the `original` stage are stashed and later used in the patch preparation. When building patches for a package from distribution the objects built during `original` stage must be compatible with those from the distro's binary package.

Assembly files resulting from the correct `original` build can be stored to speed up patch builds later on.

Building patches

Next, source code patches are applied and the build is redone. This time the `kpatch_cc` wrapper is instructed by environment variable `KPATCH_STAGE=patched` to build a special patch-containing object.

Wrapper first calls real compiler with `-S` flag to produce an assembly file for the patched version, which is stored under file name `.kpatch_${filename}.patched.s`. It then calls `kpatch_gensrc` that compares original and patched files and produces a patch-containing assembly where all the changes in the code are put in the `.kpatch`-prefixed sections while original code is left as is. This assembly is finally compiled to a patch-containing object file by calling compiler with the `-c` flag.

Linking done by the project build system carries these sections to the target binary and shared object files. During the link stage `kpatch_cc` adds `ld` argument `-q` that instructs linker to keep information about all the relocations. This is required for the Patch application to (dynamically) link patch into running binary.

Then the sanity check is done, checking that the symbols originating from the non-`kpatch` sections in the patched binary are equal to those from the original binary or its debuginfo.

The last part is postprocessing the patch-containing binaries: stripping off the original binary sections, fixing relocations and prepending the resulting ELF content with a common kpatch header. Look at Making a kpatch for details.

Project patch building

The above algorithm is implemented in two various helper scripts. The first is `kpmake` that can build patches for any project buildable via `make` and the second aims at building patches for applications and libraries coming from distribution packages `scripts/pkgbuild`.

Both are using kpatch_cc wrapper described below. It is recommended to go through Manual Patch Creation at least once.

Using `kpmake`

The `kpmake` script can be used to build patches for a project built locally via `./configure && make && make install`.

The usage is simple, just call `kpmake` with a list of source patches as arguments and `kpmake` will build the binary patches and store them to `patchroot` directory.

`kpmake` requires the following simple criteria to be met on the build system:

1. The default target SHOULD be the one that builds all the files in the project. This is by default the `all` target in most of the projects.
2. The `install` target MUST install the project deliverativites into the directory specified as `DESTDIR` environment variable. This is default for most projects. Other projects are either patched by distributions to include that target or have it under a different environment variable.
3. The `clean` target SHOULD be the one that cleans the project.

The typical usage is the following for the `configur`able project:

```
$ cd project_dir
$ KPATCH_STAGE=configure CC=kpatch_cc ./configure
$ kpmake first.patch second.patch
BUILDING ORIGINAL CODE
...
INSTALLING ORIGINAL OBJECTS INTO kpmake
...
applying patch ~/first.patch
...
applying patch ~/second.patch
...
BUILDING PATCHED CODE
...
INSTALLING PATCHED CODE
...
MAKING PATCHES
patch for foobar is in patchroot/${buildid}.patch
...
```

Available options are:

`--help, -h`	display a short help,
`--update`	just update the `kpatches`. Useful when working on the kpatch tools,
`--clean`	invoke `make clean` before building,
`--srcdir DIR`	change to the `DIR` before applying patches.

Note that `kpmake` uses `kpatch_cc` under the hood. Read about it kpatch_cc.

Building patch for a package via `scripts/pkgbuild`

The `scripts/pkgbuild` is responsible for the building of the patch and pre-building the original package and assembly files. At the moment it only supports the building of the RPM-based packages.

Each package has its own directory `packages/$distro/$package` with different package versions as subdirectories. For instance, the directory `packages/rhel7/glibc/` contains subdirectory `glibc-2.17-55.el7` that has the configuration and scripts for building and testing of the sample security patches for that version of `glibc` package for RHEL7.

The project directory contains three main files:

1. Shell-sourceable `info` that has the necessary environment variables specified along with the hooks that can alter package just before the build and test patch before it is packed. For instance, `packages/rhel7/glibc/glibc-2.17-55.el7/info` contains both hooks and a `kp_patch_test` function that runs glibc test suite with each invocation being patched with the built patch.
2. The list `plist` of the patches to be applied. File names are relative to the top-level directory `patches`.
3. YAML file `properties.yaml` containing version-specific configuration, such as URLs for pre-build storage, original source packages URL, and Docker container images with toolchain (GCC/binutils) version is required to properly build the package.

   This is not used at the moment and left as an information source for the users.

The Doctor: `kpatch_user`

All the job is done by the `kpatch_user`. It is called `doctor` hereafter and the targets of operations are thus called `patients`.

The doctor accepts a few arguments that are common for all types of operations:

`-v`	enable verbose output
`-h`	show commands list

Applying patches via `patch`

The `patch` mode patches a process with ID given as an argument to `-p` option or all of them except self and `init` when the argument is `all`. The patch (or directory with patches) to be applied should be specified as the only positional argument:

```
$ kpatch_user patch -p <PID_or_all> some_patch_file.kpatch
```

The patches are basically ELF files of relocatable type `REL` with binary meta-information such as BuildID and name of the patch target prepended. Loading patches is thus a lot like loading a shared object (library) into a process. Except we are puppeting it by strings going through a keyhole in other process' memory.

First, the memory near the original object is allocated, then all the relocations and symbols are resolved in a local copy of patch content. This pre-baked patch is copied to the patient's memory and, finally, original functions are overwritten with the unconditional jumps to the patched version.

For more details look at the chapter Patching.

Cancelling patches via `unpatch`

The `unpatch` mode makes doctor remove patches listed by target BuildID from the patients' memory. It simply restores the original code of the patched functions from a stash allocated along with the patch and puppets patients to `munmap` the memory areas used by patches.

Showing info via `info`

The last entry to the `kpatch_user` is the `info` command that lists all the objects and their BuildIDs for the set of the processes requested. Its primary use is as the utility for the book-keeping software.

Patchlevel support

Since patches to the objects such as libraries can be updated, there is a way to distinguish them, called `patchlevel`. This information is parsed from the layout of the directory where the patches are stored. If on patching stage a patch with a bigger `patchlevel` is found, the old one is removed and the new one is applied.

Where To Start Hacking

For the impatient: the code is located in the `src` directory. Module responsible for the process-start interception `binfmt` is located in its own directory. Tests are located in the `tests` directory.

Project directory

The root directory contains project-level makefile. Run:

```
$ make
```

and enjoy libcare being deployed on your machine.

After that run tests by simply emitting:

```
$ make tests
```

The following is the project subdirectories:

1. tests contains all the tests for the project and should be used as a sample for building the `kpatches`.
2. binfmt contains kcare-user specific implementation of the binary file format that overrides kernel's `elf_format` and notifies about start-up of the binaries listed using `/proc/ucare/applist`.

Test infrastructure `./tests`

This directory contains the tests and the infrastructure to run them. To keep the `tests` directory clean, each test is placed in its own directory.

To run the tests emit:

```
$ make
```

this will build and run all the tests discovered for all types of build and all flavors of the `kpatch_user` usage.

There are two types of test builds.

The first one is the regular build done by manually emitting assembler files for both original and patched source files, and then applying `kpatch_gensrc` to them and compiling the result into a kpatch-containing object where from it was extracted from by the utils, as described in Manual Patch Creation section.

The second one is the build done by the `kpmake` tool which uses `kpatch_cc` compiler wrapper, as described in kpmake section. The build results for each build type are placed in their own subfolder ina test directory.

A test can be built with the particular build type using either `make build-$test` or `make kpmake-$test` commands.

Sometimes it is necessary to debug a particular test so all changes MUST retain the ability to run the tests manually. The manual run is done by executing an appropriate binary (with the `LD_LIBRARY_PATH` set as needed) and target `kpatch_user patch` at its process.

However, it is recommended to run tests by the `./run_tests.sh` script, available in the `tests` directory.

The `run_tests.sh` script accepts the following options:

`-f FLAVOR`	execute `FLAVOR` of tests from those listed in test flavors.
`-d DESTDIR`	assume that test binaries are located in `DESTDIR` subdirectory of a test. The `build` subdirectory is a default one. Use `kpmake` to run the tests build with the kpmake with binaries stored in the subdirectory with the same name.
`-v`	be verbose

The only argument it accepts is a string with space separated names of tests to execute. The default is to execute all the tests discovered.

Test flavors

There are the following test flavors. Most of the tests are executed in all flavors, it depends on what `should_skip` function of `run_tests.sh` returns. Some of the tests have different success criteria between different flavors: e.g. `fail_*` tests check that binary is succesfully patched upon execution with `test_patch_startup` flavor.

The flavors are:

`test_patch_files`

(default) that simply executes a test process and points `kpatch_ctl patch` to it, doing so for present patches for both binary and shared libraries.

`test_patch_dir`

that executes a test and patches it with a per-test patch-containing directory fed to `kpatch_ctl patch`.

`test_patch_startup`

that starts a `kcare_genl_sink` helper that listens to notifications about a start of a listed binary and executes `kpatch_ctl patch` with the directory containing patches for all the tests discovered.

`test_patch_patchlevel`

that checks that patchlevel code works as expected. This applies two patches with different patch levels to the `patchlevel` test and checks that the patching is done to the latest one.

Adding or fixing a test

Each test has its own directory that MUST have the file named `desc` which contains a one-line description of the test. The `desc` files are used to discover the tests.

The makefile inside the test directory MUST compile the code into a binary. The binary name MUST coincide with the directory and test name, the library name (if present) must be equal to `lib$test.so`. The source code is typically called `$test.c` for the binary and `lib$test.c` for the library. Patch files are `$test.diff` and `lib$test.diff`.

When the above rules are followed the test can simply include `../makefile.inc` file that will provide build system for all of the build types described above.

The `tests/makefile.inc` file itself includes either `makefile-kpmake.inc` file when the `CC` variable equals `kpatch_cc` or `makefile-patch.inc` otherwise. The former provides a set of rules that meet `kpmake`s criteria described in kpmake. The later provides a set of rules described in Manual Patch Creation, except for the libraries output that is broken with them and requires including of a makefile `makefile-patch-link.inc` that links the shared library to extract proper names of the sections for the kpatch. For the usage example take a look at the test `both` that tests patching of both binary and a library it loads.

`fastsleep.so`

To speed up test execution while allowing them to be run manually we had to adjust tests with a `LD_PRELOAD`ed library that redefines `sleep` and `nanosleep` to change their arguments so the code sleeps faster. The code is in the file `fastsleep.c`.

Intercept start of a new process by `./binfmt`

The project must be able to patch a just executed process. This is required whenever updates have not been installed or to patch a process that can dynamically load via `dlopen` one of the shared library we have a patch for.

This is implemented by a kernel module that inserts a handler for binary file format `binfmt` overriding the default one for the ELF file. The task of the `binfmt` is just to wrap the original functions provided by the kernel and check whether the path of an executed binary is listed.

When it is the subscribed userspace, an application is notified by the Generic Netlink channel implemented by the kernel module. The sample application `kcare_genl_sink` provides an example on how to implement userspace counterpart for the channel. It is also used for testing.

The main module function is the `do_intercept_load` in the file `binfmt.c`.

It checks if the path of an application being executed is listed in the file `/proc/ucare/applist` and therefore the execution should be intercepted. This list should contain **real** file paths without double slashes, `.` or `..`.

To add an application write its path to `/proc/ucare/applist` file. Multiple paths can be added at once, separated by a newline character. To remove a path, write it with the minus sign prefixed. To clear the list write magic `-*` to it.

If an execution needed to be intercepted as told by the aforementioned call, the `binfmt` module tries to notify about the new process by sending a message to the subscribed process, if any. If there is no one listening on the other side, the code just leaves the binary as is, continuing its normal execution. Otherwise, we enter an infinite loop waiting for the signal `SIGSTOP` to come, blocking all the other signals, **including** `SIGKILL` and `SIGSEGV`. The `doctor` code executed by the subscribed application such as `kcare_genl_sink` that simply calls `kpatch_user patch` must attach to that newborn `patient` and apply its remedies.

Source directory `src`

The `src` directory contains libcare project code.

The following files are updated as a part of the project:

1. `src/kpatch_user.c` has the top-level code for the user-space patching it uses code from the rest of kpatch modules of kcare-user. The entry point is the `cmd_patch_user` function.
2. `src/kpatch_elf.c` contains the ELF-format specific functions such as parsing the program headers, resolving symbols, and applying relocations to the loaded patch. This uses libelf.
3. `src/kpatch_ptrace.c` implements ptrace(2) functions such as reading/writing patient's memory, executing code on the behalf of patient (e.g. syscalls), and parsing the patient's auxiliary vector to determine real entry point of the application.
4. `src/kpatch_strip.c` contains two modes of operation: `--strip` that removes all non-kpatch sections from the ELF file, and `--undo-link` that redoes binary image offsets into section offsets for symbols, relocations' offsets, and addends and resets section addresses to zero, converting an ELF object to `REL` type.

5. `src/kpatch_gensrc.c` is the powerhorse of patching. It compares original versus patched assembly files and produces an assembly file with all the changes stored into `.kpatch`-prefixed sections.

   The code is changed so all the variable access is done through the Global Offset Table entries referenced via PC-relative instructions (option `--force-gotpcrel`). The `jump table` is generated by the `kpatch_user.c` code and filled with `kpatch_elf.c` code. See below for details.

How Does It Work

It's a miracle. Really. We got somewhat lucky that all the tools were ready before we ever started working on this. Thank you, Open Source The Mighty!

Short Introduction to ELF

Most of the binaries in the system are in the elf(5) format. From the producer point of view, the file of this format consists of a set of blocks called `sections`. Sections can contain data (`.rodata`, `.data`), executable code (usually called `.text`) and auxiliary data. The text references its parts and necessary data (such as variables) by means of `symbols`. For instance, C's `main` is a special symbol where the control is transferred by the C runtime after the required initialization is done. The symbols are listed in the section `.symtab` whenever it is required.

There are three main types of ELF format files: the `DYN`amic shared object, used primarily to store common code in libraries; the binary `EXEC`utable, used to contain the application; and the `REL`ocatable object file resulting from compiling an assembler file (GNU C compiler actually generates assembler which is fed to the GNU assembler).

These differ mostly by the types of relocations they may and do contain. Relocations are the technique used to allow address changes in the binary object files. For instance, when linking a set of `.o` files into executable, the linker merges sections from them into a single section such as `.text`, `.data` or `.rodata`. The linker then adjusts relocation info such as the place where it should be applied (called `r_offset`), target symbol and its address and/or addend relative to the symbol value (called `r_addend`). Some types of relocations are also allowed in the final binary object and are resolved upon load by the dynamic linker.

The `DYN`amic object contains all the data necessary to load the library on a random base address. This randomization of the base leads to randomization of the library functions addresses, making it harder for an intruder to exploit a vulnerability, and allowing multiple libraries to be loaded without interfering each other. Because it is impossible to know the address of a variable at the compile time the `DYN`amic code refers to its data objects using so-called Global Offset Table (GOT). This table contains addresses of the variables, so accessing a variable takes two steps: first loading the GOT entry, then unreferencing it. GOT entry is usually referenced in the instruction pointer relative manner. The GOT is filled by the dynamic linker such as `ld-linux` while resolving relocations from the `.rela.dyn`. Only a few types of relocations are allowed there, they are (for x86-64): `R_X86_64_RELATIVE`, `R_X86_64_64` and `R_X86_64_GLOB_DATA`. The symbols provided by the `DYN`amic object are listed in the `.dynsym` section with the names stored in the `.dynstr` section. Special section `.dynamic` contains all the data required to load an object, such as a list of required libraries, pointers to the relocation entries and so on.

The `EXEC`utable objects are usually linked to a fixed address and contain no relocation information. The kernel only needs to know how to load this type of objects along with the interpreter if specified. Most of the binaries have the dynamic linker `ld-linux` specified as the interpreter. It is loaded by the kernel and the control is transferred here. The dynamic loader duty is to load all the necessary libraries, resolve symbols and transfer the control to the application code.

The `REL`ocatable object file can contain any type of relocation. The static linker, such as `ld`, links these into an `EXEC`utable file or a `DYN`amic one. The `REL` object file is merely an assembler file turned into a binary file, with the symbol references noted as appropriately. That is, for every symbol reference in the assembler file there is a corresponding symbol added to the `REL`ocatable ELF file and the relocation referencing this symbol. For every symbol defined the corresponding symbol is added to the `.symtab` section. ASCII zero-ended string names are stored into the `.strtab` section. The static linker then resolves symbol referenced in one object file with the symbols defined in another object file or DYNamic shared object file.

Patching

Here we are going to describe how the patching is performed.

This is the act that looks like a mix of static and dynamic linking in the process address space expecting that we are doing it using `ptrace`. There is infant task to reuse `rtld`'s `_dl_open` calls to do the job for us.

The following is the verbose description of the `kpatch_process_patches` function flow.

Attaching

When a user asks `kpatch_user` to patch a process with a given patch (or a directory with patches), the patcher (let's call it `doctor`) first attaches to the threads to be patched (let's call it `patient`) thus stopping their execution.

Execute Until Libraries Are Loaded

Now, if we are about to patch a freshly executed binary, we have to continue its execution until all the libraries are loaded. That is, if the binary has a non-zero `interpreter`, such as `ld-linux`, the kernel first executes the interpreter and it is the interpreter task to transfer control to the application text after all the initialization is successful. So, to ensure that all the libraries are loaded so we can use symbols provided by them in our patches, we have to wait until the initialization is done. We do this by inserting a `TRAP` instruction at the entry point of the application, so when the interpreter is done loading the libraries, we have to parse auxiliary vector information to find the entry point. This is done in the `kpatch_load_libraries` function.

Examine Application Object Files

The next step is to find out what ELF objects are loaded and where. This way we know offsets for reach dynamically-loaded library and can actually resolve symbols from there. This is done by the function `kpatch_create_object_files`. For the correct mapping of the object symbol addresses to the virtual address space we have to parse the instructions on how to load the object stored in the `program headers` part of the ELF, and are used by the dynamic loader or the kernel. This part is done by the function `kpatch_create_object_files`.

Locate Patches For Objects

Next, if we are given a patch file we check that there are indeed patches for the objects of the application (`kpatch_verify_objects`). If we are given a directory, we lookup for patch files named `$BuildID.kpatch` inside it and load what we have found (`kpatch_load_patches`). If there are no patches we just let the application continue its execution, free our resources and we are done.

Applying Patches

Otherwise, we call the `kpatch_apply_patches` function that goes through the list of objects that do have patches and applies patches.

Regular executable objects (both `EXEC` and `DYN`) reference global functions via Procedure Linkage Table and global object symbols by copying them into local data using `R_X86_64_COPY` relocation (for `EXEC`) or looking for the address in the application or library using `R_X86_64_GLOB_DATA` relocation (for `DYN`). We had to implement a jump table for the function references which is reused as GOT for the symbol reference. It is also used as the Thread Pointer Offset table for the TLS data.

So, the first we need to count if there is a need for the jump table at all. For that, we do count undefined and TLS symbols and allocate the jump table if there are any of them.

The next we need to find a region in the patient address space suitable to mmap the patch here. We start to look for the hole after the object and check if there is enough space to fit the patch, looking farther upon failure. This is done by the `kpatch_find_patch_region` function.

We allocate an anonymous region to hold the patch on the patient's behalf using the code injected by `ptrace`. This is done by the `kpatch_mmap_remote` function that executes a `mmap` syscall remotely.

Once we got the address of the region and allocated memory there, we are all prepared to resolve the relocations from the kpatch.

Applying Relocations

Resolving symbols

Since relocations are made against symbols we have first to resolve symbols. This is done by the function `kpatch_resolve` present in the `kpatch_elf.c` file.

We resolve sections addresses first. We know the address of the region we allocated for the `kpatch`, so we can calculate the `kpatch`'s sections addresses. Other sections' addresses are resolved from the original object file we are about to patch.

After the section addresses are resolved we resolve addresses for the symbols present in the kpatch. The functions and data objects symbols of type `STT_FUNC` and `STT_OBJECT` have the containing section offset added to the `st_value`.

The thread-local storage objects of type `STT_TLS` may be referenced by two different relocations, one that gets offset from a GOT (`GOTTPOFF`), another that asks offset to be put inline (`TPOFF{32,64}`). We use symbol field `st_size` to store the original offset and `st_value` to store the offset in the jump table.

Objects of unknown type `STT_NOTYPE` are resolved via the jump table. If it is later discovered that they are referenced by a relocation as a Global Offset Table entry such as `GOTPCREL` then only the address value from the jump table is used.

Rest of the symbol types are unsupported. The appearence of the unsupported symbol type will cause the `doctor` to fail.

Doing relocations

Now that we are all set, we resolve the relocations. This is done by the function `kpatch_relocate` that calls `kpatch_apply_relocate_add` for all the sections of type `SHT_RELA`.

The code is pretty straightforward except for two relocations. The first one is the `TPOFF{32,64}` relocations that do restore offset saved in `st_size`. Another one is Global Offset Table-related relocations such as `GOTTPOFF`, `GOTPCREL`, and Ubuntu Xenial specific `REX_GOTPCRELX`. If the referenced symbol has type `STT_NOTYPE` or `STT_TLS`, then the jump table entry is reused as the Global Offset Table entry. If the relocation aims for either original object or patch section, then we convert the `mov` instruction present to the `lea` instruction as there is no appropriate jump table entry which is not required in that case since the target section is closer than 2GiB (we allocate the memory for the patch that way).

Doctor injects the patch

Now that the patch is fully prepared it is written into the previously allocated region of patient's memory.

But we are not yet done with the patching of the patient. We now have to reroute the execution paths from the old buggy functions into our just loaded new shiny ones. But it is dangerous to patch functions that are being executed at the moment, since this can change the way the data is structured and corrupt everything. So, we have to wait until the patient leaves functions we are about to patch.

This is done by the function `kpatch_ensure_safety` which checks that there is no patched symbols on the stack and, if there is any, waits for the patient to hit breakpoints placed at their returns. The function uses `libunwind` function with pluggable unwinder interfaces.

If we ensured the patching safety, we start the patching itself. For that the entry point of the original functions are rewritten with the unconditional jumps to the patched functions. This is done by the function `kpatch_apply_hunk` called for each of the original functions that do have patched one.

Doctor exits

At this point doctor done with his job, it frees resources and leaves. If anything wrong happens during any of the actions the appropriate error MUST be printed.

Manual Patch Creation

Throughout this section the availability of the kpatch tools is assumed. To build them and add them into PATH, do:

```
$ make -C src
$ export PATH=$PWD/src:$PATH
```

Generating the kpatch assembler with `kpatch_gensrc`

So, the main working horse for the whole project, including kernel patches, is the `kpatch_gensrc` utility. It compares two assembler files and whenever there are differences in the code of a particular function, it emits a new code after the original one but with a name suffixed with `.kpatch` and in the `.kpatch.text` section. Keeping the original code maintains all the data and references in the original order. All the new variables are being put into `.kpatch.data` section.

So, imagine that you have two source code versions available, let's name them `foo` for the original and `bar` for the patched version:

```
// foo.c
#include <stdio.h>
#include <time.h>

void i_m_being_patched(void)
{
    printf("i'm unpatched!\n");
}

int main(void)
{
    while (1) {
        i_m_being_patched();
        sleep(1);
    }
}
```

```
// bar.c
#include <stdio.h>
#include <time.h>

void i_m_being_patched(void)
{
    printf("you patched my %s\n", "tralala");
}

int main(void)
{
```

```
    while (1) {
        i_m_being_patched();
        sleep(1);
    }
}
```

Now we need to get assembler code for both of the files:

```
$ gcc -S foo.c
$ gcc -S bar.c
```

Take a look at what is different in the files:

```
$ diff -u foo.s bar.s
--- foo.s   2016-07-16 16:09:16.635239145 +0300
+++ bar.s   2016-07-16 16:10:43.035575542 +0300
@@ -1,7 +1,9 @@
-   .file   "foo.c"
+   .file   "bar.c"
    .section    .rodata
 .LC0:
-   .string "i'm unpatched!"
+   .string "tralala"
+.LC1:
+   .string "you patched my %s\n"
    .text
    .globl  i_m_being_patched
    .type   i_m_being_patched, @function
@@ -13,8 +15,10 @@
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
-   movl    $.LC0, %edi
-   call    puts
+   movl    $.LC0, %esi
+   movl    $.LC1, %edi
+   movl    $0, %eax
+   call    printf
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
```

You can see that the GCC optimized a call to a `printf` without arguments to a simple `puts` call, and our patch brings the `printf` call back.

Now it's time to produce a patch result. Execute `kpatch_gensrc`:

```
$ $KPATCH_PATH/kpatch_gensrc --os=rhel6 -i foo.s -i bar.s -o foobar.s
FATAL! Blocks of type other mismatch 1-1 vs. 1-1
```

Oops, the difference in `.file` is fatal. Let's trick that and try again:

```
$ sed -i 's/bar.c/foo.c/' bar.s
$ $KPATCH_PATH/kpatch_gensrc --os=rhel6 -i foo.s -i bar.s -o foobar.s
```

The result is:

```
    .file   "foo.c"
#---------- var ---------
    .section    .rodata
.LC0:
    .string "i'm unpatched!"
#---------- func ---------
    .text
    .globl  i_m_being_patched
    .type   i_m_being_patched, @function
i_m_being_patched:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %edi
    call    puts
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   i_m_being_patched, .-i_m_being_patched
i_m_being_patched.Lfe:
#---------- kpatch begin ---------
    .pushsection .kpatch.text,"ax",@progbits
    .globl  i_m_being_patched.kpatch
    .type   i_m_being_patched.kpatch, @function
i_m_being_patched.kpatch:
.LFB0.kpatch:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0.kpatch, %esi
    movl    $.LC1.kpatch, %edi
    movl    $0, %eax
    call    printf
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0.kpatch:
    .size   i_m_being_patched.kpatch, .-i_m_being_patched.kpatch
i_m_being_patched.kpatch_end:
    .popsection

    .pushsection .kpatch.strtab,"a",@progbits
kpatch_strtab1:
    .string "i_m_being_patched.kpatch"
```

```
	.popsection
	.pushsection .kpatch.info,"a",@progbits
i_m_being_patched.Lpi:
	.quad i_m_being_patched
	.quad i_m_being_patched.Lfe - i_m_being_patched
	.quad i_m_being_patched.kpatch
	.quad i_m_being_patched.kpatch_end - i_m_being_patched.kpatch
	.quad kpatch_strtab1
	.quad 0
	.popsection

#---------- kpatch end -----------
#---------- func ---------
	.globl	main
	.type	main, @function
main:
.LFB1:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
.L3:
	call	i_m_being_patched
	movl	$1, %edi
	movl	$0, %eax
	call	sleep
	jmp .L3
	.cfi_endproc
.LFE1:
	.size	main, .-main
	.pushsection .kpatch.data,"aw",@progbits
.LC0.kpatch:
	.string "tralala"
	.popsection
	.pushsection .kpatch.data,"aw",@progbits
.LC1.kpatch:
	.string "you patched my %s\n"
	.popsection
	.ident	"GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
	.section	.note.GNU-stack,"",@progbits
```

A watchful reader have spotted two new sections: `.kpatch.info` and `.kpatch.strtab`. The former contains information about the function being patched and the patch itself, such as sizes of the functions. The compiler generates a relocation section `.rela.kpatch.info` against it that references symbols from both the original binary as patch targets and the patch as the patched function.

We should now compile both original and patched assembler files into binaries, keeping the relocation information with linker's `-q` switch:

```
$ gcc -o foo foo.s
$ gcc -o foobar foobar.s -Wl,-q
```

and proceed to the building a `kpatch` file out of these.

Making a kpatch

Removing non-kpatch sections with `kpatch_strip --strip`

The binary containing patch (`foobar` in the example above) has extra sections:

```
$ readelf -S foobar | grep -A 1 kpatch
  [16] .kpatch.text      PROGBITS         0000000000400662  00000662
       000000000000001a  0000000000000000  AX       0     0     1
  [17] .rela.kpatch.text RELA             0000000000000000  00001ef0
       0000000000000048  0000000000000018          40    16     8
--
  [20] .kpatch.strtab    PROGBITS         000000000040069b  0000069b
       0000000000000019  0000000000000000   A       0     0     1
  [21] .kpatch.info      PROGBITS         00000000004006b4  000006b4
       0000000000000030  0000000000000000   A       0     0     1
  [22] .rela.kpatch.info RELA             0000000000000000  00001f38
       0000000000000048  0000000000000018          40    21     8
--
  [36] .kpatch.data      PROGBITS         0000000000601050  00001050
       000000000000001b  0000000000000000  WA       0     0     1
```

This is where the patch actually hides and we had to extract it from here. First, we need to strip all the unnecessary data from the patched binary:

```
$ kpatch_strip --strip foobar foobar.stripped
$ stat -c '%n: %s' foobar foobar.stripped
foobar: 10900
foobar.stripped: 6584
```

The `--strip` mode of the `kpatch_strip` operation removes all the `kpatch`-unrelated sections, setting their type to `PROG_NOBITS` and modifying sections offsets.

Fix up relocations

Patch code, packed into `.kpatch.text` section, references its part and parts of the original binary via relocations.

These relocations are fixed by invoking `kpatch_strip --rel-fixup` as follows:

1. All relocations of type `PLT32` are changed to `PC32` since they are resolved via the jump table.
2. All the relocations internal to the patch are left as is -- that is, if newly introduced code references newly introduced function or data. The `doctor` will have enough information to resolve these.
3. Some of these relocations are referencing original local symbols introduced by compiler named like `.LC0`. Each relocation referencing such a symbols is replaced to relocation referencing section that contains them with an updated `r_addend`.
4. Relocations referencing Thread Local Storage symbols are harder to handle, mostly because of the variety of TLS models in use.

   Relocations of type `TPOFF32` are generated in `EXEC`utable binaries for TLS symbols defined in application. We ensure that (negative) offset values into TLS block coincide between original and patched binaries.

   Relocations of type `GOTTPOFF` are generated when code references TLS variable from another object. These are tricky: code looks for appropriate original `GOT` entry which is filled via `TPOFF64` relocation and writes the offset of this entry into the `r_addend` field of `GOTTPOFF` relocation.

   All the other TLS relocation types are not supported since there is no full TLS support yet.

Another important part is the interaction between `kpatch_gensrc` generation of `GOTPCREL` entries and linker optimization for it.

Whenever assembly code of the patch references variable not coming from patch there are two options.

First, the referenced variable can be defined in the original code that can be referenced as is since we allocate patches close to the original code and the 32-bit PC-relative relocation should be enough.

Second, the referenced non-TLS variable can be imported by the original code, e.g. from `glibc` library. In that case, the variable can be further than 2GiB away from the patch code and it ought to have a way to address it in all the 64-bit address space.

There is no reliable way to distinguish these at the compile time, so we replace **EVERY** reference to a non-patch variable with an indirect reference using a Global Offset Table entry. This is what `--force-gotpcrel` option of `kpatch_gensrc` does.

Linker knows what symbols are defined in original binary and what symbols are coming from imported shared libraries. Linker resolves symbols coming from the original binary by setting a correct original section number to the symbol. Symbols defined in the patch are assigned section number of either `.kpatch.text` or `.kpatch.data` at this stage.

Some linker versions optimize our two-stage references to original symbols via `GOTPCREL`:

```
mov foobar@GOTPCREL(%rip), %rax
mov (%rax), %rax
```

into one-stage

```
lea foobar(%rip), %rax
mov (%rax), %rax
```

changing relocation type from `GOTPCREL` to a simple `PC32`. The `kpatch_strip` code ensures that this is always done for known symbols so there is no dependency on particular linker behavior.

All the references to the variables imported by the original code are left with the `GOTPCREL` relocation and these are correctly resolved during the patching, **except** for the variables `COPY` ed by the original binary.

Stripping extra information via `strip --strip-unneeded`

Now that we have fixed `kpatch` relocations we can finally strip all the unnecessary symbols with `strip`:

```
$ strip --strip-unneeded foobar.stripped
```

This will remove the symbols that have no relocations targeted at them, so, most of the symbols, except for the sections, patched functions with `.kpatch` suffix and symbols referenced from the patch.

Undoing offsets `kpatch_strip --undo-link`

Since the `doctor` does not care for the program section and loads patch as a single bulk region without caring for the program header and sections virtual addresses and offsets in the patch must be prepared accordingly. That means we have to undo all the offsets and convert base-address relative values into section-relative values for the relocations offsets (`r_offset`), symbols (`st_value`) and finally reset the sections addresses to zeroes (`sh_addr`). This all is done by the `--undo-link` mode of `kpatch_strip`:

```
$ readelf -rs foobar.stripped
Relocation section '.rela.kpatch.text' at offset 0x11f0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000400667  00030000000a R_X86_64_32       0000000000601050 .kpatch.data + 0
00000040066c  00030000000a R_X86_64_32       0000000000601050 .kpatch.data + 8
```

```
000000400676  002500000002 R_X86_64_PC32     0000000000000000 printf@@GLIBC_2.2.5 - 4

Relocation section '.rela.kpatch.info' at offset 0x1238 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
0000004006b4  000100000001 R_X86_64_64       00000000004004d0 .text + ed
0000004006c4  002600000001 R_X86_64_64       0000000000400662 i_m_being_patched.kpat + 0
0000004006d4  000200000001 R_X86_64_64       000000000040069b .kpatch.strtab + 0

Symbol table '.symtab' contains 39 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000004004d0     0 SECTION LOCAL  DEFAULT   14
     2: 000000000040069b     0 SECTION LOCAL  DEFAULT   20
     3: 0000000000601050     0 SECTION LOCAL  DEFAULT   36
...
    37: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    38: 0000000000400662    26 FUNC    GLOBAL DEFAULT   16 i_m_being_patched.kpatch
```

Now let's undo the link:

```
$ kpatch_strip --undo-link foobar.stripped
```

Take a look at the patch afterwards to ensure that offsets have been indeed reset:

```
$ readelf -rs foobar.stripped
Relocation section '.rela.kpatch.text' at offset 0x11f0 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000005  00030000000a R_X86_64_32       0000000000000000 .kpatch.data + 0
00000000000a  00030000000a R_X86_64_32       0000000000000000 .kpatch.data + 8
000000000014  002500000002 R_X86_64_PC32     0000000000000000 printf@@GLIBC_2.2.5 - 4

Relocation section '.rela.kpatch.info' at offset 0x1238 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  000100000001 R_X86_64_64       0000000000000000 .text + ed
000000000010  002600000001 R_X86_64_64       0000000000000000 i_m_being_patched.kpat + 0
000000000020  000200000001 R_X86_64_64       0000000000000000 .kpatch.strtab + 0

Symbol table '.symtab' contains 39 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT   14
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT   20
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT   36
...
    37: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    38: 0000000000000000    26 FUNC    GLOBAL DEFAULT   16 i_m_being_patched.kpatch
```

Adding meta-information with `kpatch_make`

Finally, we need to prepend the `kpatch` ELF object with meta-information doctor uses to check that the patch target is correct.

We do this using `kpatch_make`, but first we need to know what is the name of the target object (`foo` in our case) and what is its BuildID, stored in `.note.build-id` section:

```
$ readelf -n foo | grep 'Build ID'
    Build ID: 9e898b990912e176275b1da24c30803288095cd1
```

Now we are all set to convert `foobar.stripped` into a `kpatch`:

```
$ kpatch_make -b "9e898b990912e176275b1da24c30803288095cd1" \
  foobar.stripped -o foo.kpatch
```

Now let's apply that:

```
(terminal1) $ ./foo
i'm unpatched!
i'm unpatched!
...
(terminal2) $ kpatch_ctl -v patch -p $(pidof foo) ./foo.kpatch
...
(terminal1)
you patched my tralala
you patched my tralala
```

Conclusion

Congratulations, we are done with the simple patch! It was pretty complicated, wasn't it?

Building any real project following the recipe above is a nightmare since it requires interfering with the project's build system: changing all the compilation to go through intermediate assembly and `kpatch_gensrc`.

Luckily, this can be done in a `gcc` wrapper like kpatch_cc. It allows for the transparent compilation of the patches and hides away the details into an additional abstraction layer that will eventually break, be sure.