4 BINARY TAXONOMY

If you look down on one side, everything seems reassuringly familiar. . . .
On the other side, it seems completely alien territory.
—Mary Beard, *SPQR*

Like code review and fuzzing, reverse engineering is a topic that could fill a whole book (and does; several, in fact). Rather than examining the granular details of each discipline, this book focuses on *strategy*, marshaling limited resources effectively to attain a specific and significant objective. To achieve that goal, you need to understand the lay of the land before getting into the weeds. This allows you to focus your time and effort on technical approaches that are more likely to yield new vulnerabilities.

For reverse engineering, instead of popping your binaries into Ghidra or IDA Pro and tackling assembly code head-on, you should first learn to triage and select interesting binaries for further analysis. Not all binaries are created (or rather, compiled) equal.

In this chapter, you'll learn about three common categories of binaries: scripts, intermediate representations (IR) such as bytecode, and machine code. You will then reverse engineer examples from each category. In addition, you'll venture deeper into several subcategories of these binaries, which require different approaches.

Beyond Executable Binaries and Shared Libraries

Understanding the different types of binaries helps you select the right tools and techniques to reverse engineer them. By breaking them down into a few broad categories, you'll be able to quickly triage your target and optimize your approach.

At a high level, when we think of binaries, we usually think of two kinds: executable binaries and shared libraries. As the name suggests, executable binaries can be executed directly from the command line or user interface. Shared libraries export functions that other binaries can use via static or dynamic linking. In some cases it's also possible to execute shared libraries, such as by calling dynamic-link libraries (DLLs) on Windows with `rundll32`.

These binaries come in the Portable Executable (PE) file format for Windows, the Executable and Linkable Format (ELF) for Linux, and the Mach object (Mach-O) file format for macOS and iOS. These formats are handled natively by the underlying operating system and contain the instructions to execute the binaries, as well as additional data like import and export tables, dynamic linking information, and global variables.

While this is a straightforward way to categorize binaries, it misses a lot of important details, especially in today's modern development environment. Consider some of the most popular communication software out there, like WhatsApp, Slack, and Zoom. These applications are distributed as executable binaries, but they actually package together other formats, such as Node.js scripts, WebAssembly binary code, and Common Intermediate Language (CIL) bytecode. Unlike standard executable file formats, like PE and ELF, these formats are executed in other mediums, such as the Node.js environment or the Common Language Runtime (CLR) virtual machine used by the .NET Framework. In turn, these mediums come with their own sets of security boundaries, default protections, and potential misconfigurations.

For example, in the early years of the Electron Node.js desktop application framework, an attacker could trivially escalate a simple cross-site scripting (XSS) bug to code execution. Electron allowed developers to turn on a `nodeIntegration` setting that enabled Node.js APIs and modules in the web renderer process, which effectively disabled the browser sandbox protections. This happened despite the hard lessons developers had learned by fiddling with the browser sandbox since ActiveX and Flash. Creating a bridge between what happens in the sandbox (executing JavaScript) and on the desktop (executing operating system APIs) greatly increases the blast radius of a web vulnerability. What would've been a bug limited to a single website now becomes a full-blown remote code execution on the victim's computer.

Unfortunately, we can expect more blurring of lines as web technologies continue to seep into desktop and server-side execution environments.

From a vulnerability researcher's perspective, however, this blurring of lines opens up the range of targets to reverse engineer. Compared to pure assembly code, it's relatively easier to decompile intermediate representations like Java bytecode and CIL. In fact, with the proper metadata, you can retrieve the near-original source code of these binaries. This does not even cover scripting languages like Node.js or Python, which can be packaged into binaries that run the embedded interpreter on stored scripts. Rather than decompiling machine code, reverse engineering these types of binaries involves unpacking and sometimes deobfuscating these scripts. After that, you can simply perform code review as usual.

Additionally, there are many cross-interactions between these components. For example, a Node.js script could instantiate a WebAssembly binary module, or CIL bytecode could load unmanaged libraries. To maintain a bird's-eye view of the various paths taken by the application logic, you need to understand the different types of binaries and the most effective ways to analyze them. Let's dive in, starting with scripts.

Scripts

Script files are written in a programming language that can be executed directly by an interpreter without needing to compile a binary. Common scripting languages include JavaScript, Python, and Ruby. For example, in the Node.js environment, JavaScript scripts are executed by the V8 JavaScript engine outside of the browser.

However, this does not necessarily mean that the interpreter does not compile scripts at all. Many modern interpreters employ some form of just-in-time or ahead-of-time compilation that occurs at execution time. This compiles the script into bytecode or machine code, which is more optimized and runs faster than if the script was interpreted.

Some script-based executables may contain only the compiled bytecode instead of the original scripts. In other cases, the executables may contain scripts that are obfuscated or minified (minimized), increasing the difficulty of analyzing them. In the best-case scenario, the executable simply acts as a wrapper around the source code files and executes them with an embedded interpreter. In this section, you'll explore these scenarios through two open source projects written in scripting languages and distributed as executables: DbGate, a Node.js Electron application, and Galaxy Attack, a Python PyInstaller application.

Reverse Engineering Node.js Electron Applications

These days, you're likely to encounter at least one Node.js Electron application on a desktop environment, so it's important to understand how to reverse engineer them. One of the most significant trends in modern application development is the growth of hybrid software that blends web and native solutions. Traditionally, native software built for desktops and servers was written in compiled languages like C++. Compiled languages run much faster than interpreted languages (like JavaScript and Python) due to compile-time optimizations and the ability to execute machine code directly rather than through an interpreter.

However, the emergence of the powerful just-in-time compilation V8 engine in 2008 allowed web developers to run JavaScript with better performance. This was followed by the release of Node.js in 2009, which provided a server-side JavaScript runtime environment built on V8. Instead of running it only in the browser sandbox to add functionality to web pages, developers could now write JavaScript code to read and write files, make database queries, and execute other server-side functions.

The nonblocking, event-driven architecture of Node.js also allowed developers to easily build scalable real-time applications that could handle multiple connections simultaneously. This was an essential feature for web servers, and it was where Node.js found the most initial adoption because it meant web developers could now write web applications in JavaScript for both the frontend and the backend.

Next, the Electron framework (originally named Atom Shell, in reference to the Atom code editor that it was built for) emerged. Electron focused on creating desktop applications with Node.js and other web technologies, like HTML and CSS. Instead of struggling with various operating system–specific APIs and build processes, developers could simply use tried-and-tested common environments like Node.js and the Chromium browser engine to create cross-platform desktop applications with JavaScript. This enabled much faster development, especially as desktop applications began to rely on more and more web features.

An Electron application consists of the Electron prebuilt binary, which includes the Node.js and Chromium execution environments, and the application source code, which is usually packaged into an Atom Shell Archive (ASAR) file. You can explore this with the releases of DbGate, an open source database client built on the Electron framework. For Linux, DbGate is distributed as both a Debian package and an AppImage. Download the Debian package for version 5.2.7 at *https://github.com/dbgate/dbgate/releases/download/v5.2.7/dbgate-5.2.7-linux_amd64.deb* and use the `dpkg-deb` tool to extract it. You should see the following files:

```
$ dpkg-deb -x dbgate-5.2.7-linux_amd64.deb dbgate
$ tree --charset ascii dbgate
dbgate
|-- opt
|   `-- DbGate
|       |-- chrome_100_percent.pak
|       |-- chrome_200_percent.pak
|       |-- chrome_crashpad_handler
|       |-- chrome-sandbox
❶ |       |-- dbgate
|       |-- icudtl.dat
```

```
❷ |       |-- libEGL.so
  |       |-- libffmpeg.so
  |       |-- libGLESv2.so
  |       |-- libvk_swiftshader.so
  |       |-- libvulkan.so.1
  --snip--
  |       |-- resources
❸ |       |   |-- app.asar
  |       |   `-- app.asar.unpacked
  |       |       |-- node_modules
  |       |       |   |-- better-sqlite3
  |       |       |   |   `-- build
  |       |       |   |       `-- Release
  |       |       |   |           `-- better_sqlite3.node
  |       |       |   `-- oracledb
  |       |       |       `-- build
  |       |       |           `-- Release
  |       |       |               |-- oracledb-5.5.0-darwin-x64.node
  |       |       |               |-- oracledb-5.5.0-linux-x64.node
  |       |       |               `-- oracledb-5.5.0-win32-x64.node
  |       |       `-- packages
  |       |           `-- api
  |       |               `-- dist
  |       |                   |-- 45c2d7999105b08d7b98dd8b3c95fda3.node
  |       |                   `-- 9bf76138dc2dae138cb17ee46c4a2dd1.node
  |       |-- resources.pak
  |       |-- snapshot_blob.bin
  |       |-- swiftshader
  |       |   |-- libEGL.so
  |       |   `-- libGLESv2.so
  |       |-- v8_context_snapshot.bin
  |       `-- vk_swiftshader_icd.json
```

From the listing, you can see the package contains a *dbgate* executable binary ❶. This is simply a prebuilt Electron binary that loads the bundled ASAR package. You can also find shared libraries for graphics rendering and media parsing ❷, which are dependencies used by Chromium and Node.js. The ASAR file *app.asar* ❸ is located in the *resources* directory. Electron automatically loads the application from this directory.

This is a common pattern for not only Electron but also script-based executables. The application package will typically include a common script interpreter, some additional library files, and a script bundle. As you encounter more of these types of executables, you'll be able to recognize specific patterns, such as the presence of an ASAR file, that will tell you what kind of framework is used.

If you have Node.js installed, you can unpack the ASAR file with the asar tool:

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.0/install.sh | bash
$ source ~/.zshrc
$ nvm install --lts
$ npm install -g asar
$ npx asar extract dbgate/opt/DbGate/resources/app.asar dbgate-src
$ tree --charset ascii dbgate-src
dbgate-src
--snip--
|-- icon.png
|-- node_modules
|   |-- @yarnpkg
|   |-- argparse
--snip--
|-- package.json ❶
|-- packages
|   |-- api
|   |   `-- dist
|   |       |-- 45c2d7999105b08d7b98dd8b3c95fda3.node
|   |       |-- 9bf76138dc2dae138cb17ee46c4a2dd1.node
|   |       `-- bundle.js
|   |-- plugins
|   |   |-- dbgate-plugin-csv
|   |   |   |-- dist
|   |   |   |   |-- backend.js
|   |   |   |   `-- frontend.js
|   |   |   |-- icon.svg
|   |   |   |-- LICENSE
|   |   |   |-- package.json
|   |   |   `-- README.md
--snip--
`-- src
    |-- electron.js
    |-- mainMenuDefinition.js
    |-- nativeModulesContent.js
    `-- nativeModules.js
```

There are many interesting filenames in the unpacked code, but in most cases the first point of reference should be a manifest file that includes important metadata about the package, such as the entrypoint file that will be executed first. Different programming language packages use manifests; for Node.js, the manifest is *package.json* ❶, for Java *MANIFEST.MF*, for Go *go.mod*, and so on. Let's take a look at DbGate's *package.json*, shown in Listing 4-1.

```
{
    "name": "dbgate",
    "version": "5.2.7",
    "private": true,
    "author": "Jan Prochazka <jenasoft.database@gmail.com>",
    "description": "Opensource database administration tool",
    "dependencies": {
        "electron-log": "^4.4.1",
        "electron-updater": "^4.6.1",
        "lodash.clonedeepwith": "^4.5.0",
        "patch-package": "^6.4.7"
    },
 ❶ "repository": {
        "type": "git",
        "url": "https://github.com/dbgate/dbgate.git"
    },
    "homepage": "./",
 ❷ "main": "src/electron.js",
    "optionalDependencies": {
        "better-sqlite3": "7.6.2",
        "oracledb": "^5.5.0"
    }
}
```

Listing 4-1: The DbGate manifest file

There are two useful pieces of information here. First, the manifest tells you where the original source code repository is ❶, which would be invaluable if you encountered this binary without knowing it was open source. Second, it tells you that the entrypoint denoted by `main` is *src/electron.js* ❷. This would be the next file to investigate.

You're making good progress, but before long you may encounter the following obstacle in *electron.js*:

```
if (!apiLoaded) {
    const apiPackage = path.join(
        __dirname,
        process.env.DEVMODE ? '../../packages/api/src/index' : '../packages/api/dist/
            bundle.js' ❶
    );

    global.API_PACKAGE = apiPackage;
    global.NATIVE_MODULES = path.join(__dirname, 'nativeModules');

    // console.log('global.API_PACKAGE', global.API_PACKAGE);
    const api = require(apiPackage);
```

The code does import a package from *packages/api/dist/bundle.js* in production ❶, but if you inspect this file, it's a mess of tightly packed code and obscure variable names, making it impossible to manually analyze.

This is because DbGate uses Webpack and Rollup, module bundlers for JavaScript that combine various source code files into one or more minified output files that are more optimized for distribution. In the original DbGate source code, you can find the Webpack configuration files at *packages/api/webpack.config.js* and the Rollup configuration file at *packages/web/rollup.config.js*. To go any further, you'll need to somehow reverse the minification.

Unpacking Source Maps

Due to the minified output, it's usually impossible to recover the original unpacked version of the code from a Webpack or Rollup output file. However, in some cases developers may configure these tools (and others, like Babel and TypeScript) to also output a source map file. JavaScript source maps are special files that map transformed source code files like minified Webpack output to the original source code, including the original directory structure. This enables easier debugging of JavaScript code during development.

In the case of DbGate, the developer has not enabled source maps for Webpack but has done so for two Rollup output files, *query-parser-worker.js* and *bundle.js*, as shown in Listing 4-2.

rollup.config.js

```
export default [
    {
        input: 'src/query/QueryParserWorker.js',
        output: {
          ❶ sourcemap: true,
            format: 'iife',
          ❷ file: 'public/build/query-parser-worker.js',
        },
        plugins: [
            commonjs(),
            resolve({
                browser: true,
            }),

            // If we're building for production (npm run build
            // instead of npm run dev), minify
            production && terser(),
        ],
    },

    {
        input: 'src/main.ts',
        output: {
            sourcemap: true,
            format: 'iife',
            name: 'app',
            file: 'public/build/bundle.js',
        },
```

Listing 4-2: The Rollup configuration that enables source maps

The `sourcemap` value ❶ tells you that Rollup will include a source map when generating the output file at the indicated path ❷.

In the extracted files for the DbGate package, *bundle.js* and *bundle.js.map* can be found in the same directory, *packages/web/public/build*. Take a moment to compare the two files. While *bundle.js* appears to be JavaScript code, it's highly minified and difficult to read. Meanwhile, *bundle.js.map* appears to be a JSON file with recognizable filepaths and source code.

Thanks to the source map file, you can convert *bundle.js* from an incomprehensible blob of code into the actual source code files. Use Mozilla's `source-map` library to quickly write a script to do so. Place *bundle.js.map* and the *unpack.js* file, whose code is shown in Listing 4-3, in the same directory (this file is also available in the book's code repository, at *chapter-04/unpack-sourcemap*).

unpack.js

```
const fs = require('fs');
const path = require('path');
const sourceMap = require('source-map');

const rawSourceMap = JSON.parse(fs.readFileSync('bundle.js.map', 'utf8'));

fs.mkdirSync('output');

sourceMap.SourceMapConsumer.with(rawSourceMap, null, consumer => {
 ❶ consumer.eachMapping(mapping => {
        const sourceFilePath = mapping.source;
        const sourceContent = consumer.sourceContentFor(mapping.source);

        // Remove path traversal characters
      ❷ const normalizedSourceFilePath = path
            .normalize(sourceFilePath)
            .replace(/^(\.\.(\/|\\|$))+/, '');
        const outputFilePath = path.join('output', normalizedSourceFilePath);
        const outputDir = path.dirname(outputFilePath);

        if (!fs.existsSync(outputDir)) {
            fs.mkdirSync(outputDir, { recursive: true });
        }
      ❸ fs.writeFileSync(outputFilePath, sourceContent, 'utf8');
    });
});
```

Listing 4-3: A source map unpacking script

The script parses the source map and iterates through each mapping ❶, extracting the filepath and content of the mapping. However, if you examine *bundle.js.map*, you'll notice that some source filepaths are relative paths. Unfortunately, this means we lose some information about the actual directory structure of the source code. Because we're unable to reconstruct the relative paths, we must instead treat them as being in the same root directory by removing any relative paths ❷. Nevertheless, the most important information, the contents of the source code files, is preserved and written to the output ❸.

Install the `source-map` library and run the script, which should take a few minutes:

```
$ npm install source-map
$ node unpack.js
```

Compare the output folder with the original source code. As discussed, the directory structure is not a perfect match, but it closely follows *packages/web/src* in the original source code. Additionally, you may notice that TypeScript files, like *packages/filterparser/src/getFilterType.ts*, have been converted into JavaScript files, like *filterparser/lib/getFilterType.js*. This is because TypeScript is actually *transpiled* (meaning compiled to a different programming language) to JavaScript during the build process, so it can be interpreted by JavaScript engines. Observe some of the differences between the original TypeScript in Listing 4-4 and the transpiled JavaScript in Listing 4-5.

```
❶ import { isTypeNumber, isTypeString, isTypeLogical,
      isTypeDateTime } from 'dbgate-tools';
  import { FilterType } from './types';

❷ export function getFilterType(dataType: string): FilterType {
      if (!dataType) return 'string';
      if (isTypeNumber(dataType)) return 'number';
      if (isTypeString(dataType)) return 'string';
      if (isTypeLogical(dataType)) return 'logical';
      if (isTypeDateTime(dataType)) return 'datetime';
      return 'string';
  }
```

Listing 4-4: The original `getFilterType` code

In the original TypeScript, the source code uses the `import` keyword to import dependencies ❶, but this is supported only in newer versions of JavaScript, such as ECMAScript 6. In addition, it includes type annotations that specify variable types ❷, which are not natively supported in JavaScript.

getFilterType.js

```
  "use strict";
  Object.defineProperty(exports, "__esModule", { value: true });
  exports.getFilterType = void 0;
❶ const dbgate_tools_1 = require("dbgate-tools");
❷ function getFilterType(dataType) {
    if (!dataType)
        return 'string';
    if ((0, dbgate_tools_1.isTypeNumber)(dataType))
        return 'number';
    if ((0, dbgate_tools_1.isTypeString)(dataType))
        return 'string';
    if ((0, dbgate_tools_1.isTypeLogical)(dataType))
        return 'logical';
    if ((0, dbgate_tools_1.isTypeDateTime)(dataType))
        return 'datetime';
    return 'string';
}
exports.getFilterType = getFilterType;
```

Listing 4-5: The converted getFilterType code

In contrast, the transpiled JavaScript uses a backward-compatible CommonJS standard `require` keyword to import dependencies ❶, and it drops the type annotations ❷ (these will have been checked at the transpilation stage). This loses some information that could speed up reverse engineering, since type declarations add details about the expected inputs. For example, in the original source code, *packages/filterparser/src/types.ts* tells you that `FilterType` should be one of the following strings:

```
// import types from 'dbgate-types';

export type FilterType = 'number' | 'string' | 'datetime' | 'logical' |
    'eval' | 'mongo';
```

While there don't appear to be any other major differences that would significantly affect your analysis of the code, you must take the increased verbosity of transpiled JavaScript into account as well. As you encounter more transpiled or transformed (such as minified) code, you'll learn to map common patterns in transpiled JavaScript back to their TypeScript equivalents, such as boilerplate export code or *polyfills* (code that implements functions that are natively supported in newer versions of JavaScript but not in older versions).

In cases where the TypeScript hasn't been transpiled to JavaScript, you may still notice some subtle differences. For example, study the original code from *packages/web/src/clientAuth.ts* in Listing 4-6.

clientAuth.ts

```
  import { apiCall, enableApi } from './utility/api';
  import { getConfig } from './utility/metadataLoaders';
  --snip--
❶ export async function handleAuthOnStartup(config) {
      if (config.oauth) {
          console.log('OAUTH callback URL:', location.origin
                      + location.pathname);
      }
```

```
    if (config.oauth || config.isLoginForm) {
        if (localStorage.getItem('accessToken')) {
            return;
        }

        redirectToLogin(config);
    }
}
```

Listing 4-6: The original `handleAuthOnStartup` *code*

The code uses the `async` keyword to define an asynchronous function ❶. Asynchronous functions return a `Promise` that allows the program to call the function but continue executing and responding to other events. In comparison, the code *output/src/clientAuth.ts* in Listing 4-7 looks somewhat different.

clientAuth.ts

```
import { __awaiter } from "tslib";
--snip--
export function handleAuthOnStartup(config) {
 ❶ return __awaiter(this, void 0, void 0, function* () {
        if (config.oauth) {
            console.log('OAUTH callback URL:', location.origin +
                location.pathname);
        }
        if (config.oauth || config.isLoginForm) {
            if (localStorage.getItem('accessToken')) {
                return;
            }
            redirectToLogin(config);
        }
    });
}
```

Listing 4-7: The converted `handleAuthOnStartup` *code*

Instead of `async`, the converted code uses the TypeScript `__awaiter` polyfill function, which provides the same features as an asynchronous function ❶.

Like with the transpiled JavaScript, these differences should not pose a significant challenge. However, we're still losing some directory structure information. For example, the extracted code does not include the *packages* or *web* directories. This can hamper your efforts to analyze the application code, as you can't confirm the exact locations of the files in relation to one another. Keep this in mind if you encounter any source maps that include directory traversal paths.

We're also missing information about non-core files, including test and configuration files. In a typical code review scenario, these files can provide additional clues about the software, such as how it was compiled.

Overall, the presence of source maps doesn't automatically mean you can retrieve the original source code. They usually bundle together only the relevant components in the codebase and lose information in the process.

For example, as we've just seen, in the case of DbGate the source map includes only the client-side code covered by the Rollup configuration, and some useful information is lost during transpilation. Still, they're handy tools if you have them. Without a source map, you must instead rely on less accurate means of reconstructing the original code, such as code beautifiers.

Using Beautifiers on Minified Code

A *beautifier* is a tool that formats code to be more human-readable, such as by adding consistent spacing and newlines. This makes it easier to analyze minified code, which by definition compresses the code as much as possible (such as by removing unnecessary spaces and newlines that an interpreter doesn't need to parse the code).

Returning to the extracted app source code archive files, you can find a different *bundle.js* file in *packages/api/dist*. Unlike the bundle file in *packages/web/public/build*, it doesn't come with a source map file to help you unpack it further. If you refer to the Webpack configuration for this bundle in the original source code at *packages/api/webpack.config.js*, you can see that the developer commented out an option that would have disabled minimization:

```
    // optimization: {
    //   minimize: false,
    // },
```

The same goes for the rest of the plug-in distribution files in *packages/plugins*. Webpack optimized the output bundles, including shortening variable and function names, removing whitespace, and eliminating dead code, resulting in a compact but seemingly undecipherable blob. Nevertheless, if you peer closely at the code, you may be able to make out a few intelligible strings and function names. This is because Webpack preserves some constant values and exported function names.

You can improve the readability of the code by using a beautifier to reformat and partially deobfuscate it. While there are several options available, the `js-beautify` package should suffice. Install the package and run it on the main bundle, using the following commands:

```
$ npm -g install js-beautify
$ npx js-beautify packages/api/dist/bundle.js > bundle.beautified.js
```

The beautified code reveals a fairly consistent structure of a list of function definitions. You may even spot code similar to the files you unpacked using the source map earlier, because the server- and client-side code share some common imported functions. One of these is `compileMacroFunction`:

```
function compileMacroFunction(macro, errors = []) {
    if (!macro) return null;
    let func;
    try {
     ❶ return func = eval(getMacroFunction[macro.type](macro.code)), func
    } catch (e) {
        return errors.push(`Error compiling  macro ${macro.name}:
        ${e.message}`), null
    }
}
```

Notice the dangerous `eval` sink ❶, which executes its string argument as JavaScript. This could become an easy code injection vulnerability if the argument can be controlled by an attacker. Since Webpack does not obfuscate standard function names like `eval` by default, you can run automated code analysis tools to quickly flag such dangerous sinks in beautified code, especially when it's difficult to manually review it yourself.

Analyzing a Dangerous Sink

Since `compileMacroFunction` appears in both the frontend and backend code and includes a dangerous sink, it's worth digging into. Using the techniques you learned in the previous chapters, you can analyze the unpacked and beautified code to figure out whether it's an exploitable vulnerability.

The function first takes a `macro` argument that is passed to `getMacroFunction`, and the result of this is finally passed to `eval`. Let's take a look at the code for `getMacroFunction` from the unpacked source map:

```
const getMacroFunction = {
 ❶ transformValue: code => `
(value, args, modules, rowIndex, row, columnName) => {
    ${code}
}
`,
 ❷ transformRow: code => `
(row, args, modules, rowIndex, columns) => {
 ❸ ${code}
}
`,
};
```

From the code, you can see that `getMacroFunction` is actually an object literal with only two keys, `transformValue` ❶ and `transformRow` ❷. The values of these keys are functions that take a single argument interpolated within a string that defines another function. Recall that this string is eventually passed to `eval`.

As such, it appears that as long as an attacker can control `macro.code`, they have a good chance of triggering a code injection. Now you can work backward using the sink-to-source analysis approach.

In the beautified and the unpacked backend code, `compileMacroFunction` is called in the `runMacroOnChangeSet` function:

```
function runMacroOnChangeSet(
 ❶ macro,
    macroArgs,
    selectedCells,
    changeSet,
    display,
    useRowIndexInsteaOfCondition
) {
    var _a;
    const errors = [];
  ❷ const compiledMacroFunc = compileMacroFunction(macro, errors);
```

The function takes a `macro` argument ❶ that is eventually passed to the `compileMacroFunction` function without any modifications ❷. However, if you search for `runMacroOnChangeSet` in the beautified code, you won't get any results, meaning a sink-to-source path doesn't exist. If you search the unpacked code, you'll find that it's called in a few *.svelte* files, which are used as part of the Svelte frontend framework to define frontend components. For example, it's used in *TableDataGrid.svelte*:

```
❶ function handleRunMacro(macro, params, cells) {
  ❷ const newChangeSet = runMacroOnChangeSet(macro, params, cells,
    changeSetState?.value, display, false);
    if (newChangeSet) {
        dispatchChangeSet({ type: 'set', value: newChangeSet });
    }
}

    \$: reference = config.reference;
    \$: childConfig = config.childConfig;
</script>

<VerticalSplitter isSplitter={!!reference}>
    <svelte:fragment slot="1">
        <DataGrid
            {...\$\$props}
            gridCoreComponent={SqlDataGridCore}
            formViewComponent={SqlFormView}
            {display}
            showReferences
            showMacros
            hasMultiColumnFilter
            onRunMacro={handleRunMacro} ❸
```

Here, the frontend component defines a `handleRunMacro` function that takes a `macro` argument ❶, which is passed directly to `runMacroOnChangeSet` ❷. This function is triggered by the `onRunMacro` handler ❸, which is triggered when the user runs the macro from the frontend by clicking a button.

This appears to be a viable sink-to-source path, but it isn't a particularly exciting one. After all, if a user needs to enter the macro payload and click a button themselves to actually trigger this, then it's more like a self-inflicted code execution requiring significant user interaction. Nevertheless, this might be a good place to start digging deeper for similar vulnerable code patterns.

Reverse Engineering a Python Application

In addition to Node.js Electron applications, applications in other programming languages, such as Python and Ruby, can also be bundled into executables. After all, one big advantage of scripting languages is portability; you need only a compatible interpreter to run most scripts on any platform. Electron applications are the most common, but it's still useful to understand how to unpack some of these other types of applications, such as PyInstaller executables.

PyInstaller allows developers to bundle Python applications into a single package, such as a single-file executable. After executing the binary, PyInstaller starts a bootloader that unpacks compiled Python scripts (*.pyc*) and native libraries before running the main script with the bundled Python interpreter. This package is constructed in a fairly standard way, including a Table of Contents list and archive files.

Generally, the compressed archive data appended to the end of the executable contains the following:

- The Python dynamic library, including the interpreter
- The main Python script
- The Python zip application archive (usually named *PYZ-00.pyz*), containing additional Python scripts
- Library files
- Supporting files such as media assets

Similar to other bundled script-based executables, you can usually identify a PyInstaller executable by reviewing the strings or headers:

```
$ strings main.exe | grep pyinstaller
xpyinstaller-4.7.dist-info\COPYING.txt
xpyinstaller-4.7.dist-info\INSTALLER
xpyinstaller-4.7.dist-info\METADATA
xpyinstaller-4.7.dist-info\RECORD
xpyinstaller-4.7.dist-info\REQUESTED
xpyinstaller-4.7.dist-info\WHEEL
xpyinstaller-4.7.dist-info\entry_points.txt
xpyinstaller-4.7.dist-info\top_level.txt
$ strings ~/Downloads/main.exe | grep python
bpython310.dll
6python310.dll
```

You can confirm whether it's a PyInstaller executable by using PyInstaller's built-in `pyi-archive_viewer` utility to examine the CArchive of PyInstaller-bundled executables. For this section, we'll experiment using Amegma Galaxy Attack, a simple PyInstaller game with a Windows executable. Download *main.exe* and the source code from the GitHub release page (*https://github.com/Amegma/Galaxy-Attack/releases/tag/v1.3.0*). Next, install PyInstaller and run the archive viewer utility:

```
$ pip install pyinstaller
$ pyinstaller -v
6.8.0
$ pyi-archive_viewer main.exe
pos, length, uncompressed, iscompressed, type, name
[(0, 217, 287, 1, 'm', 'struct'),
 (217, 1018, 1754, 1, 'm', 'pyimod01_os_path'),
 (1235, 4098, 8869, 1, 'm', 'pyimod02_archive'),
 (5333, 7116, 16898, 1, 'm', 'pyimod03_importers'),
 (12449, 1493, 3105, 1, 'm', 'pyimod04_ctypes'),
 (13942, 833, 1372, 1, 's', 'pyiboot01_bootstrap'),
 (14775, 466, 696, 1, 's', 'pyi_rth_inspect'),
 (15241, 698, 1067, 1, 's', 'pyi_rth_pkgutil'),
 (15939, 1187, 2154, 1, 's', 'pyi_rth_multiprocessing'),
 (17126, 1999, 4202, 1, 's', 'pyi_rth_pkgres'),
 (19125, 2103, 3574, 1, 's', 'main'),
--snip--
❶ (5175013, 1985630, 4471024, 1, 'b', 'python310.dll'),
 (7160643, 13440, 25320, 1, 'b', 'select.pyd'),
 (7174083, 405123, 1117936, 1, 'b', 'unicodedata.pyd'),
 (7579206, 56136, 108544, 1, 'b', 'zlib1.dll'),
--snip--
 (38446628, 12, 4, 1, 'x', 'pyinstaller-4.7.dist-info\\INSTALLER'),
 (38446640, 2714, 7085, 1, 'x', 'pyinstaller-4.7.dist-info\\METADATA'),
 (38449354, 13562, 56668, 1, 'x', 'pyinstaller-4.7.dist-info\\RECORD'),
 (38462916, 8, 0, 1, 'x', 'pyinstaller-4.7.dist-info\\REQUESTED'),
 (38462924, 104, 98, 1, 'x', 'pyinstaller-4.7.dist-info\\WHEEL'),
 (38463028, 141, 361, 1, 'x', 'pyinstaller-4.7.dist-info\\entry_points.txt'),
 (38463169, 20, 12, 1, 'x', 'pyinstaller-4.7.dist-info\\top_level.txt'),
❷ (38463189, 2076778, 2076778, 0, 'z', 'PYZ-00.pyz')]
```

Partway down the list you'll find `python310.dll` ❶, which tells you that the version of Python used in this release by PyInstaller was version 3.10. However, other than `main` and the media assets, there don't appear to be any source code files. This is because they're packed into the *PYZ-00.pyz* ❷ ZlibArchive file, which you can examine in the interactive session:

```
? O PYZ-00.pyz
Contents of 'PYZ-00.pyz' (PYZ):
 is_package, position, length, name
 0, 17, 1893, '__future__'
 0, 1910, 1651, '_aix_support'
 0, 3561, 1388, '_bootsubprocess'
 0, 4949, 2937, '_compat_pickle'
 0, 7886, 2213, '_compression'
 0, 10099, 5991, '_osx_support'
 0, 16090, 2422, '_py_abc'
 0, 18512, 51188, '_pydecimal'
 0, 69700, 7845, '_strptime'
 0, 77545, 2863, '_threading_local'
 0, 80408, 25050, 'argparse'
 0, 105458, 22331, 'ast'
 1, 127789, 453, 'asyncio'
```

You'll find that some module names match the original source code files, while the rest come from imported support modules. Extract `models.button`, then exit the `pyi-archive_viewer` interactive session:

```
? X models.button
to filename? models.button.pyc
? q
```

The extracted file is a compiled Python file. If you view the contents of the file, you'll encounter mostly gibberish. This is because compiled Python files consist of bytecode instead of the original source code. This runs faster because it allows the Python interpreter to skip parsing the plaintext code and run lower-level instructions with more optimizations.

However, when you extract the compiled bytecode directly, you lose the starting magic bytes of the ZlibArchive file. These correspond to the release version of Python (consisting of 2 bytes) followed by the carriage return and line feed characters (`0D0A`). The version is important because each new Python version makes changes to the interpreter that affect the structure of the compiled bytecode, which affects how it should be decompiled.

These magic bytes are missing because PyInstaller stores a single instance of them near the start of the *PYZ-00.pyz* ZlibArchive file containing the compressed *.pyc* files. For example, the first 16 bytes of *PYZ-00.pyz* are `50595A00 6F0D0D0A 001F8838 00000000`. The first 4 bytes, representing the ASCII string `PYZ`, are followed by the magic bytes you need: `6F0D0D0A`.

Prepend these bytes followed by 12 null bytes of padding to *models.button.pyc*:

```
$ echo -n -e '\x6F\x0D\x0D\x0A' > fixed.models.button.pyc
$ printf '\x00%.0s' {1..12}  >> fixed.models.button.pyc
$ cat models.button.pyc >> fixed.models.button.pyc
```

After extracting and preparing the compiled Python file, you need to actually decompile it. Among the various open source decompilers, Decompyle++ tries to support bytecode from any version of Python, which is helpful since Galaxy Attack is compiled in a later version. Clone and build Decompyle++, then run it on the modified compiled Python file:

```
$ git clone https://github.com/zrax/pycdc
$ cd pycdc
$ cmake .
$ make
$ make check
$ cd ..
$ pycdc/pycdc fixed.models.button.pyc
```

If you performed the steps correctly, you should get fairly coherent output, as shown in Listing 4-8.

```
# Source Generated with Decompyle++
# File: fixed.models.button.pyc (Python 3.10)

import pygame
from utils.assets import Assets
from config import config
from constants import Font, Colors

class Button:

    def __init__(self, color, outline_color, text = ('',)):
        self.color = color
        self.outline_color = outline_color
        self.text = text
        self.outline = False
        self.rect = pygame.Rect(0, 0, 0, 0)

    def draw(self, pos, size):
        self.default_outline = pygame.Rect(pos[0] - 5, pos[1] - 5, size[0] + 10, size[1] + 10)
        self.on_over_outline = pygame.Rect(pos[0] - 6, pos[1] - 6, size[0] + 12, size[1] + 12)
        self.rect = self.default_outline
        default_inner_rect = (pos[0], pos[1], size[0], size[1])
        onover_inner_rect = (pos[0] + 1, pos[1] + 1, size[0] - 2, size[1] - 2)
        inner_rect = onover_inner_rect if self.outline == True else default_inner_rect
        pygame.draw.rect(config.CANVAS, self.outline_color, self.on_over_outline \
            if self.outline == True else self.default_outline, 0, 7)
        pygame.draw.rect(config.CANVAS, self.color, inner_rect, 0, 6)
        if self.text != '':
            font = pygame.font.Font(Font.neue_font, 40)
            Assets.text.draw(self.text, font, Colors.WHITE, \
                (pos[0] + size[0] / 2, pos[1] + size[1] / 2), True, True)
            return None ❶
    def isOver(self):
        return self.rect.collidepoint(pygame.mouse.get_pos())
```

Listing 4-8: The decompiled Button class

If you compare the output to the original source code file *models/button.py*, you'll find that there are only minor differences (such as an additional `return None` ❶) in the decompiled code. By repeating this process for the other compiled Python files extracted from the PyInstaller executable, you should be able to retrieve near-original source code.

Although actual software distributed as PyInstaller executables is much rarer than Electron applications, working on this simple example helps illustrate some common patterns in reverse engineering software written in scripting languages. It's impossible to completely remove the presence of source code, even if it's been compiled, transpiled, or bundled in some way. However, the degree of lossiness can have a significant impact on the ease of analysis.

Intermediate Representations

In terms of abstraction, intermediate representations lie between machine code and the source code. As the name suggests, these are higher-level representations of source code that can be interpreted and executed by a runtime.

There are several advantages to using an intermediate representation. For example, the runtime can take over many routine tasks, such as memory management, garbage collection, and exception handling, which can free developers to focus on simply building the application without needing to add all of this in their source code. Intermediate representations can also make it easier for runtimes to perform type checking or debugging, which makes a program more robust.

Although compiled Python bytecode can be considered a form of intermediate representation, it operates differently from the C# and Java intermediate representations you'll be analyzing in this section. Python bytecode is compiled at a higher level of abstraction than C# and Java, which makes it easier to retrieve the original source code. While reverse engineering script-based binaries focuses on extraction and retrieval, reverse engineering intermediate representation binaries focuses on decompilation and reconstruction.

Although Python bytecode is still executed by the Python interpreter, Java and C# (or rather, .NET) binaries are executed in their respective virtual machine runtime environments. A Java class file should be able to run in any operating system as long as a compatible Java virtual machine (JVM) is available. This makes it easier to reverse engineer than machine code compiled binaries, which target specific instruction sets and architectures.

Finally, another characteristic of intermediate representation binaries is that they usually include additional metadata that affects their runtime environment configuration. For example, the Java Archive (JAR) package file format includes a manifest that tells the JVM which class corresponds to the application entry point, what dependencies are required, and other important information. Similarly, .NET binaries, also known as *assemblies*, include a manifest containing metadata such as version numbers, included files, and references. Assemblies also include metadata about every type and member they use, which is extremely useful for decompilation.

Intermediate representations are important to identify because that will allow you to apply a more straightforward means of reverse engineering and decompilation that provides more accurate output. The information preserved in terms of the expected argument types, classes, and variables is invaluable, and it can save you hours of analysis. However, you may also encounter a major challenge with obfuscation explicitly meant to prevent reverse engineering. This might force you to apply dynamic analysis strategies (which we will explore in the next chapter).

As in the previous section, we'll explore reverse engineering of intermediate representation binaries through two open source examples. In keeping with the theme of the last set of examples, they're also a database client and a game, respectively. For C#, you'll work on LiteDB Studio, and for Java, you'll tackle Pixel Wheels.

Common Language Runtime Assemblies

.NET is an open source developer platform for building applications written in C#, F#, and Visual Basic. The key foundation of .NET is the Common Language Runtime, which runs Common Intermediate Language intermediate representation instructions. To actually execute the code, the CLR converts the CIL to processor-specific instructions using just-in-time or ahead-of-time compilation.

.NET binaries are distributed as assemblies, which can take the form of either *.exe* or *.dll* files. The assembly format is essentially an extension of the Portable Executable format and is encapsulated within the standard PE structure. After the PE headers, the binary contains CLR-specific data:

Assembly manifest Assembly metadata

Type metadata Metadata tables that define the types and members used in the assembly

CIL code The actual intermediate language code that is executed in the CLR

Resources Assets such as images, configuration, and other data

Strong name signature An optional digital signature to verify the assembly

You can explore this by analyzing LiteDB Studio, a graphical interface for viewing and editing LiteDB database files. Since the executable was compiled for Windows and the tools used to reverse engineer it are primarily Windows-based, you should perform the steps described here on Windows if possible. If that's not an option, it is possible to run the tools on other platforms, with varying degrees of difficulty.

Download the LiteDB Studio binary from *https://github.com/mbdavid/LiteDB.Studio/releases/download/v1.0.3/LiteDB.Studio.exe*. You can use the PE-Bear tool to view some of the properties of the assembly; download the latest release from *https://github.com/hasherezade/pe-bear/releases*.

As the name suggests, PE-Bear parses and disassembles PE files, and it even handles .NET assemblies. As well as the standard PE headers, you should see a .NET Hdr tab in the main window, which corresponds to the assembly manifest. Within that tab, you can view CLR-specific metadata such as `MajorRuntimeVersion`, the virtual addresses, and the sizes of the other metadata streams, including `Metadata` (type metadata), `Resources`, and `StrongName Signature`. The virtual address and size of `StrongNameSignature` are 0, which means there is no strong name signature set for this assembly.

It's important to note the .NET header starting in the `.text` section of the PE file after the standard PE headers, which reinforces the fact that .NET assemblies are actually an extension of the PE file format. If you check the value of the `.text` raw address in the Section Hdrs tab, you'll see that it matches with the first offset in the .NET Hdr tab. However, you can't analyze the .NET headers much further with PE-Bear.

Viewing the hex dumps of the `Metadata` or `Resources` streams reveals a few familiar-looking strings and a lot of non-ASCII bytes. For example, the start of the metadata table looks like this:

```
00000000  42 53 4a 42 01 00 01 00 00 00 00 00 0c 00 00 00  |BSJB............|
00000010  76 34 2e 30 2e 33 30 33 31 39 00 00 00 00 05 00  |v4.0.30319......|
00000020  6c 00 00 00 5c ba 02 00 23 53 74 72 69 6e 67 73  |l...\º..#Strings|
00000030  00 00 00 00 c8 ba 02 00 24 2b 02 00 23 55 53 00  |....Èº..$+..#US.|
00000040  ec e5 04 00 d6 3a 02 00 23 42 6c 6f 62 00 00 00  |ìå..Ö:..#Blob...|
00000050  c4 20 07 00 10 00 00 00 23 47 55 49 44 00 00 00  |Ä ......#GUID...|
00000060  d4 20 07 00 c8 4a 08 00 23 7e 00 00 00 49 6d 6d  |Ô ..ÈJ..#~...Imm|
00000070  47 65 74 44 65 66 61 75 6c 74 49 4d 45 57 6e 64  |GetDefaultIMEWnd|
00000080  00 53 65 6e 64 4d 65 73 73 61 67 65 00 43 72 65  |.SendMessage.Cre|
```

These bytes need to be parsed in a manner specific to the .NET assembly format. Rather than doing this manually, you can turn to tools that do it for you. As mentioned previously, several different high-level programming languages can compile to CIL, a bytecode language interpreted by the CLR. CIL is an object-oriented and stack-based instruction set that is not dependent on a specific processor. You can disassemble any .NET assembly into CIL using the IL Disassembler tool that comes with Visual Studio.

If you haven't already installed Visual Studio on Windows, install it with the .NET Framework tools to access the IL Disassembler. Once installed, you should be able to run it with `ildasm.exe` in the Visual Studio Developer Command Prompt. As a quick test, compile the C# code in Listing 4-9 in Visual Studio using the Console App (.NET Framework) template.

Program.cs

```
using System;

public class Hello
{
    public static void Main(String[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Listing 4-9: A sample .NET Framework program

Check the output pane in Visual Studio to determine the location of the build output. Next, open the Visual Studio Developer Command Prompt and disassemble the file with IL Disassembler:

```
> ildasm.exe /out=disassembled.il C:\repos\ConsoleApp1\ConsoleApp1\bin\Debug\
ConsoleApp1.exe
```

The disassembled CIL file should look similar to this truncated output:

```
// Metadata version: v4.0.30319
.assembly extern mscorlib ❶
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
.assembly ConsoleApp1 ❷
{
--snip--
}
.module ConsoleApp1.exe ❸
// MVID: {796768DC-788B-4A50-85E3-0615D98C7C6D}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00020003    //  ILONLY 32BITPREFERRED
// Image base: 0x00000274A3D40000

// ============== CLASS MEMBERS DECLARATION ===================

.class public auto ansi beforefieldinit Hello ❹
    extends [System.Runtime]System.Object
{
    .method public hidebysig static void  Main(string[] args) cil managed ❺
    {
        .entrypoint
        .custom instance void System.Runtime.CompilerServices.
            NullableContextAttribute::.ctor(uint8) = ( 01 00 01 00 00 )
        // Code size       11 (0xb)
        .maxstack  8
        IL_0000:  ldstr      "Hello World!"
        IL_0005:  call       void [System.Console]System.Console::
            WriteLine(string)
        IL_000a:  ret
    } // end of method Hello::Main

    .method public hidebysig specialname rtspecialname ❻
        instance void  .ctor() cil managed
    {
        // Code size       7 (0x7)
    .maxstack  8
        IL_0000:  ldarg.0
        IL_0001:  call       instance void [System.Runtime]System.Object::.ctor()
        IL_0006:  ret
    } // end of method Hello::.ctor

} // end of class Hello
```

The CIL starts with external assembly declarations ❶. Notice the use of the `.publickeytoken` directive to uniquely identify imported assemblies by their strong name and ensure the correct version is used. Next, the actual assembly is declared ❷. This is followed by the module declaration ❸, which includes important attributes like the image base address and the application environment.

The actual class declaration ❹ includes the method declaration for the `Main` function you defined ❺ and the implicit constructor method ❻. The actual CIL instructions seem fairly straightforward, with operations such as `ldstr` and `call`. However, as you progress to more complex applications, like LiteDB Studio, it won't be as easy to read this by yourself.

If you run `ildasm.exe` without the `/out` parameter, you'll open a graphical user interface that represents the assembly in a tree. This is too rudimentary for extended reverse engineering. Instead, you can switch to ILSpy, an open source .NET assembly decompiler. Download the latest installer at *https://github.com/icsharpcode/ILSpy/releases* and open the LiteDB Studio *.exe* file with it.

ILSpy automatically parses the .NET headers and outputs the information in the initial screen when you load the assembly:

```
// C:\Users\Default\Downloads\LiteDB.Studio.exe
// LiteDB.Studio, Version=1.0.3.0, Culture=neutral, PublicKeyToken=null
// Global type: <Module>
// Entry point: LiteDB.Studio.Program.Main ❶
// Architecture: AnyCPU (32-bit preferred)
// Runtime: v4.0.30319
// Hash algorithm: SHA1

using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;

[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: AssemblyTitle("LiteDB.Studio")]
[assembly: AssemblyDescription("A GUI tool for LiteDB v5")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("LiteDB")]
[assembly: AssemblyProduct("LiteDB.Studio")]
[assembly: AssemblyCopyright("MIT")]
[assembly: AssemblyTrademark("")]
[assembly: Guid("0002e0ff-c91f-4b8b-b29b-2a477e184408")]
[assembly: AssemblyFileVersion("1.0.3.0")]
[assembly: TargetFramework(".NETFramework,Version=v4.7.2", FrameworkDisplayName = ".NET
Framework 4.7.2")]
[assembly: ComVisible(false)]
[assembly: AssemblyVersion("1.0.3.0")]
```

A key piece of information here is the entry point ❶, which you can click in ILSpy to bring you to the decompiled method.

One of the most useful features of ILSpy is the Analyze function, which you can access by right-clicking any member name. This brings up a tree containing the other members that use or are used by it, which is especially useful for sink-to-source analysis. For example, if you identify `LiteDB.Studio.MainForm.ExecuteSql` as a potential vulnerable sink, you can use the Analyze feature to find out that it's used by five other methods. You can then follow the nested Used By tree until you reach a suitable ancestor.

Of course, you aren't restricted to ILSpy's user interface. You can also right-click the assembly in the left sidebar and select **Save Code** to export the decompiled source code. From there, you can run automated code analysis tools or perform manual code review. You can also open the source code in an IDE that will provide similar analysis tools to ILSpy. Other decompilers, like JetBrains, dotPeek, and dnSpyEx, also come with debuggers to perform dynamic analysis of .NET assemblies.

Java Bytecode

Similar to the .NET Framework's CIL, Java also uses an intermediate representation that gets executed by a common runtime—in this case, the JVM platform. Like CIL, Java bytecode uses a higher-level instruction set than machine code, but unlike CIL, Java bytecode also uses registers in the form of a local variable array. In practice, you'll most often encounter Java binaries distributed as Java Archive files with the *.jar* extension.

Like .NET assemblies, JAR files bundle together bytecode (Java class files), resources, and metadata into a single file. However, while the PE format encapsulates the .NET assemblies, JAR files are simply ZIP files that can be unpacked with any archive extraction program. This can make executing them a little less intuitive, as they must be run with the Java executable instead of executing them directly.

You can observe some of the differences between CIL and Java bytecode by adapting the "Hello World" sample program to Java, as in Listing 4-10.

Hello.java

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Listing 4-10: A sample Java program

Install the Java Development Kit (JDK) and compile the program into a Java class file before running it:

```
$ sudo apt install default-jdk
$ javac Hello.java
$ java Hello
Hello World!
```

Next, run Java's built-in disassembler with flags to show all classes and members along with some additional stack information:

```
$ javap -p -v Hello.class
Classfile Hello.class
    Last modified 30 May 2023; size 416 bytes
    SHA-256 checksum 4f0ee00df8e3ff6d3cdf8cac7ad765819369ee1602b15e9a2a2b67076fb36e44
    Compiled from "Hello.java"
class Hello ❶
    minor version: 0
    major version: 63
    flags: (0x0020) ACC_SUPER
    this_class: #21                         // Hello
    super_class: #2                         // java/lang/Object
    interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool: ❷
     #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
     #2 = Class              #4             // java/lang/Object
     #3 = NameAndType        #5:#6          // "<init>":()V
     #4 = Utf8               java/lang/Object
     #5 = Utf8               <init>
     #6 = Utf8               ()V
```

```
   #7 = Fieldref           #8.#9          // java/lang/System.out:Ljava/io/PrintStream;
   #8 = Class              #10            // java/lang/System
   #9 = NameAndType        #11:#12        // out:Ljava/io/PrintStream;
  #10 = Utf8               java/lang/System
  #11 = Utf8               out
  #12 = Utf8               Ljava/io/PrintStream;
  #13 = String             #14            // Hello World!
  #14 = Utf8               Hello World!
  #15 = Methodref          #16.#17        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = Class              #18            // java/io/PrintStream
  #17 = NameAndType        #19:#20        // println:(Ljava/lang/String;)V
  #18 = Utf8               java/io/PrintStream
  #19 = Utf8               println
  #20 = Utf8               (Ljava/lang/String;)V
  #21 = Class              #22            // Hello
  #22 = Utf8               Hello
  #23 = Utf8               Code
  #24 = Utf8               LineNumberTable
  #25 = Utf8               main
  #26 = Utf8               ([Ljava/lang/String;)V
  #27 = Utf8               SourceFile
  #28 = Utf8               Hello.java
{
    Hello();
        descriptor: ()V
        flags: (0x0000)
        Code:
            stack=1, locals=1, args_size=1
                0: aload_0
                1: invokespecial #1                 // Method java/lang/Object."<init>":()V
                4: return
            LineNumberTable:
                line 1: 0

  public static void main(java.lang.String[]); ❸
      descriptor: ([Ljava/lang/String;)V
      flags: (0x0009) ACC_PUBLIC, ACC_STATIC
      Code:
          stack=2, locals=1, args_size=1
               0: getstatic     #7
               3: ldc           #13
               5: invokevirtual #15
               8: return
        LineNumberTable:
              line 3: 0
              line 4: 8
}
SourceFile: "Hello.java"
```

Along with the actual bytecode instructions, the class file contains additional information, including class metadata ❶, the constants pool ❷, and methods ❸.

By analyzing the metadata and the instructions, decompilers can approximate the original source code. Some information gets lost when compiling from source code to the intermediate representation, so decompiling from the intermediate representation back to source code may not be an exact match. For example, rather than importing variables from other classes, an intermediate representation may include the resolved value of the variable directly.

You can confirm this by reverse engineering Pixel Wheels, a top-down racing game written in Java and distributed for Linux, macOS, Windows, and Android. Download *pixelwheels-0.24.2-linux64.zip* from *https://github.com/agateau/pixelwheels/releases/tag/0.24.2*. After unzipping it, you will find the *pixelwheels* binary along with a *pixelwheels.jar* file. As in the PyInstaller example we looked at earlier, simply running `strings` on the binary will give you some big hints:

```
$ strings pixelwheels
--snip--
/lib/server/libjvm.so ❶
/lib/amd64/server/libjvm.so
/lib/i386/server/libjvm.so
JNI_GetDefaultJavaVMInitArgs
JNI_CreateJavaVM
/proc/self/exe
*Z4mainEUlSt8functionIFPvS0_EERK14JavaVMInitArgsE_
void sajson::value::assert_type(sajson::type) const
/storage/gitlab-runner/builds/HVzmC8hq/0/NimblyGames/packr/PackrLauncher/src/main/headers/
    sajson.h ❷
Error: failed to create Java VM!
```

There are several Java-related strings here that strongly suggest this binary may just be a wrapper around the JAR file ❶. In addition, there's an interesting reference to "PackrLauncher" ❷, which a quick search reveals to be a native executable packager for JAR files (*https://github.com/libgdx/packr*), confirming that you should focus your efforts on the JAR file instead.

First, select a decompiler for the Java binary. There are several free or open source options available, such as IntelliJ IDEA's Fernflower, Procyon, and JD-GUI. While Fernflower is more up to date than JD-GUI, the latter (as the name suggests) comes with a user interface that allows you to quickly explore relationships between the various classes and members, similar to ILSpy. Fernflower and Procyon are command line tools, so you'll need to explore the output in a separate Java IDE (like IntelliJ IDEA) to get this functionality.

For now, as you're comparing only the output of the decompilers with the original source code, you can use Fernflower. Get it by going to *https://mvnrepository.com/artifact/com.jetbrains.intellij.java/java-decompiler-engine*, selecting the latest release, then downloading the associated JAR file.

Place the decompiler JAR file (rename it to *java-decompiler-engine.jar*) and *pixelwheels.jar* in the same directory, then perform the decompilation with the following commands:

```
$ mkdir output
$ java -jar java-decompiler-engine.jar pixelwheels.jar output/
```

After a few seconds, a new *pixelwheels.jar* file will be created in the output directory. Unzip it to get the decompiled source code.

It may be difficult to know where to start. There are multiple resource files and directories, like *musics*, while Java files appear in different directories, such as *com* and *javazoom*.

A good place to begin is by checking the manifest file at *META-INF/MANIFEST.MF*, which tells you that the main class is `com.agateau.pixelwheels.desktop.DesktopLauncher`. This leads you to the matching Java source code file at *com/agateau/pixelwheels/desktop/DesktopLauncher.java*. You now have a convenient entry point for your analysis of the decompiled source code.

The decompiled output matches quite closely with the original source code, which you can retrieve from the release page. For example, other than comments and extra whitespace, the only significant difference between the two in *DesktopLauncher.java* is the use of imported constant values.

To observe how much information is lost when compiling to an intermediate representation and subsequently decompiling it, take a look at the original source code for *DesktopLauncher.java*. In particular, look at the code for the `setupLogging` function, shown in Listing 4-11.

```
private static void setupLogging(PwGame game) {
    String cacheDir = FileUtils.getDesktopCacheDir();
    File file = new File(cacheDir);
    if (!file.isDirectory() && !file.mkdirs()) {
        System.err.println(
            StringUtils.format(
                "Can't create cache dir %s, won't be able to log to a file", cacheDir));
        return;
    }
    String logFilePath = cacheDir + File.separator + Constants.LOG_FILENAME; ❶
    LogFilePrinter printer = new LogFilePrinter(logFilePath, Constants.LOG_MAX_SIZE);
    NLog.addPrinter(printer);
    NLog.addPrinter(new SystemErrPrinter());

    game.setLogExporter(new DesktopLogExporter(printer));
}
```

Listing 4-11: The original setupLogging code

In this original source code, `Constants.LOG_FILENAME` is imported and used when constructing the `logFilePath` string ❶. Now take a look at the decompiled source code in Listing 4-12.

```
private static void setupLogging(PwGame game) {
    String cacheDir = FileUtils.getDesktopCacheDir();
    File file = new File(cacheDir);
    if (!file.isDirectory() && !file.mkdirs()) {
        System.err.println(StringUtils.format(
            "Can't create cache dir %s, won't be able to log to a file", cacheDir));
    } else {
        String logFilePath = cacheDir + File.separator + "pixelwheels.log"; ❶
        LogFilePrinter printer = new LogFilePrinter(logFilePath, 1048576L);
        NLog.addPrinter(printer);
        NLog.addPrinter(new SystemErrPrinter());
        game.setLogExporter(new DesktopLogExporter(printer));
    }
}
```

Listing 4-12: The decompiled setupLogging code

Instead of importing a variable from `Constants`, the code uses a string literal, `"pixelwheels.log"` ❶. As part of the optimization process during compilation to Java bytecode, it appears that the imported variable was resolved and placed in the local constant pool. You can confirm this fact by decompiling *com/agateau/pixelwheels/desktop/DesktopLauncher.class* using `javap`:

```
private static void setupLogging(com.agateau.pixelwheels.PwGame);
    descriptor: (Lcom/agateau/pixelwheels/PwGame;)V
    flags: (0x000a) ACC_PRIVATE, ACC_STATIC
    Code:
        stack=6, locals=5, args_size=1
            0: invokestatic  #23
            3: astore_1
            4: new           #24                 // Class java/io/File
            7: dup
            8: aload_1
            9: invokespecial #25                 // Method java/io/File."<init>":
                                                    (Ljava/lang/String;)V
           12: astore_2
           13: aload_2
           14: invokevirtual #26                 // Method java/io/File.isDirectory:()Z
           17: ifne          47
           20: aload_2
           21: invokevirtual #27                 // Method java/io/File.mkdirs:()Z ❶
           24: ifne          47
           --snip--
```

```
64: ldc           #38                 // String pixelwheels.log ❷
66: invokevirtual #35                 // Method java/lang/StringBuilder.append:
                                         (Ljava/lang/String;)Ljava/lang/
                                         StringBuilder;
```

Without particular expertise in reading Java bytecode, you can still match up the output to the original source code. For example, the `File.mkdirs` method is invoked followed by an `ifne` condition jump instruction ❶, which corresponds to the `if-else` conditional in the source code. Eventually, the `constant .log` string is loaded onto the stack from the constant pool with `ldc #38` ❷, meaning the string has index 38 in the pool, and called with the `StringBuilder .append` method.

After decompiling the source code, you can analyze it by applying the code review strategies you learned in the previous chapters, with the caveat that you shouldn't always take the decompiled output at face value. For example, if you perform attack surface analysis on the code, you may notice an interesting `RemoteInput` class in *com/badlogic/gdx/input/RemoteInput.java* that opens the default port 8190. However, this code is not actually used elsewhere in the application, possibly because the developer decided not to enable the remote play feature.

Machine Code

Machine code is the lowest-level abstraction among the three binary categories explored in this chapter. Like binaries in general, even machine code binaries are not created or compiled equally. Programming languages such as C++, Golang, and Rust compile to machine code in different ways, and these differences can significantly affect the ease of reverse engineering them.

For now, instead of working with actual software written by other developers, you can explore these differences up close by tweaking various compiler settings yourself.

I've mentioned machine code a few times, but what exactly is it? *Machine code* consists of binary instructions that can be executed directly by the CPU and are dependent on the CPU's instruction set. An important point to remember is that machine code is not the same as assembly code. Assembly code is a human-readable, or plaintext, representation of machine code. Given the close relationship between machine code and assembly, you'll often rely on assembly language to reverse engineer binaries that have been compiled to machine code, since it's no longer possible to decompile them to the original source code files.

By matching common patterns in machine and assembly code, it's possible to convert them to *pseudocode*, which is a higher-level approximation of what the actual source code could have looked like. While pseudocode is a best-guess estimate that can be very unreliable, for simple routines, it suffices to guide your analysis.

To quickly compare machine code, assembly, and pseudocode, you can analyze a "Hello World" program written in C:

hello-world.c

```
#include <stdio.h>

int main() {
    printf("hello world\n");
    return 0;
}
```

First, compile this program with gcc:

```
$ gcc hello-world.c -o hello-world
```

Next, use the `objdump -D <FILE>` command in Linux (you can use `otool -tvV <FILE>` in macOS or `dumpbin /disasm <FILE>` in Windows) to disassemble the machine code:

```
$ gcc hello-world.c -o hello-world
$ objdump -D hello-world

hello-world:     file format elf64-x86-64
--snip--
Disassembly of section .text:

0000000000400526 <main>:
    400526: 55                  push   %rbp
    400527: 48 89 e5            mov    %rsp,%rbp
    40052a: bf c4 05 40 00      mov    $0x4005c4,%edi
    40052f: e8 cc fe ff ff      callq  400400 <puts@plt>
    400534: b8 00 00 00 00      mov    $0x0,%eax
    400539: 5d                  pop    %rbp
    40053a: c3                  retq
    40053b: 0f 1f 44 00 00      nopl   0x0(%rax,%rax,1)
```

The output may vary depending on the operating system and CPU architecture you compiled the binary for. However, it should follow the same pattern of the virtual address, the hex representation of the machine code, and the corresponding assembly instruction.

Next, download and install the Ghidra software reverse engineering framework from *https://github.com/NationalSecurityAgency/ghidra* or by running **`sudo apt-get install -y ghidra`**. Create a new project and analyze the binary with the CodeBrowser tool. In the right-hand pane, CodeBrowser will output the following pseudocode:

```
undefined8 main(void)

{
 ❶ puts("hello world");
```

```
    return 0;
}
```

You may notice that instead of `printf`, the pseudocode uses `puts` ❶. This is not a mistake by Ghidra; if you refer to the disassembled machine code, the binary actually uses `puts`. This is a compiler optimization by `gcc` that automatically converts instances of `printf` to the less resource-intensive equivalent `puts` (see *https://github.com/gcc-mirror/gcc/blob/061c331/gcc/gimple-fold.c#L3230* for the exact code that does this).

When reverse engineering such binaries, you'll be toggling regularly between the text and graph views of assembly code and pseudocode. You'll also reference metadata, if any, that is sometimes compiled into these binaries, depending on the compiler options.

In the next sections, we'll quickly examine how different compilation methods affect the difficulty of reverse-engineering machine code binaries.

Statically Linked

A *statically linked* binary is compiled with all the libraries it uses, instead of loading external libraries from the system at runtime. There are several advantages and disadvantages to this approach. On one hand, it makes the binary portable, since it can be executed independently without depending on external libraries to be installed on the operating system. On the other hand, it creates a much larger binary because more machine code must be included in the output.

You can test this with a Golang implementation of "Hello World," since Golang compiles statically linked binaries by default:

hello-world.go

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

Install Go and compile the program to a Linux x86-64 executable binary:

```
$ sudo apt install golang
$ GOARCH=amd64 GOOS=linux go build hello-world.go
$ ./hello-world
hello world
$ file hello-world
hello-world: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
   linked
```

The binary is statically linked. If you disassemble it with `objdump`, you'll get a large output because the binary contains the instructions for every single imported function. Furthermore, if you try to list the dynamic symbol table, you'll get no results because there are no dynamically linked functions. Instead, you need to dump the whole symbol table to get the statically linked functions that are now part of the binary:

```
$ objdump -t hello-world
hello-world: file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS* 0000000000000000 go.go
0000000000401000 l     F .text 0000000000000000 runtime.text
00000000004021a0 l     F .text 000000000000022d cmpbody
0000000000402400 l     F .text 000000000000013e memeqbody
0000000000402580 l     F .text 0000000000000117 indexbytebody
000000000045a760 l     F .text 0000000000000040 gogo
000000000045a7a0 l     F .text 0000000000000035 callRet
000000000045a7e0 l     F .text 000000000000002f gosave_systemstack_switch
000000000045a820 l     F .text 000000000000000d setg_gcc
--snip--
000000000047b9a0 g     F .text 0000000000000042 fmt.glob..func1
000000000047ba00 g     F .text 0000000000000092 fmt.newPrinter
000000000047baa0 g     F .text 000000000000011a fmt.(*pp).free
000000000047bbc0 g     F .text 000000000000010a fmt.(*pp).Write
000000000047bce0 g     F .text 00000000000000e5 fmt.Fprintln
```

As well as `fmt.Fprintln`, many other Golang packages and functions are included in the final binary. While the Golang linker attempts to remove dead code and unused symbols, it still needs to statically link many `fmt` package functions. If you use the Ghidra CodeBrowser to generate pseudocode for the `main` function, you'll get something similar to the following:

```
void main.main(void)
{
    long unaff_R14;
    undefined local_18 [16];

    while (&stack0x00000000 < *(undefined **)(ulong *)(unaff_R14 + 0x10) ||
           &stack0x00000000 == *(undefined **)(ulong *)(unaff_R14 + 0x10)) {
        runtime.morestack_noctxt.abi0();
    }
    local_18._8_8_ = &PTR_DAT_004b71c8;
    local_18._0_8_ = &DAT_004893e0;
 ❶ fmt.Fprintln(1,1,&PTR_DAT_004b71c8,local_18);
    return;
}
```

While the binary does the exact same thing as the C "Hello World" example, the machine code produced by the Golang compiler is harder for Ghidra to decipher. This is because Go binaries are compiled with the Go runtime, which performs additional functions such as garbage collection and stack management. Additionally, you may notice that the final output includes `fmt.Fprintln` instead of `Println` ❶. This is because in the `fmt` package, `Println` is a wrapper around the `Fprintln` function, so the compiler optimizes it away, similar to what happened with `printf` and `puts` earlier.

Dynamically Linked

In contrast to statically linked binaries, *dynamically linked* binaries are compiled with information about the libraries they depend on, but not the actual libraries themselves. The operating system parses this information and loads the libraries in memory at runtime. As a quick comparison, check the dynamic symbol table for the C "Hello World" binary using the dynamic symbol option:

```
$ objdump -T hello-world

hello-world:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 puts
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 __libc_start_main
0000000000000000  w   D  *UND* 0000000000000000              __gmon_start__
```

In Ghidra, you could click `fmt.Fprintln` to jump to the instructions for the `Fprintln` implementation, and so on. Clicking `puts` leads to an artificial "thunk function" that is meant to represent the externally loaded `puts` function at runtime:

```
0060103f                 ??         ??
         //
         // EXTERNAL
         // NOTE: This block is artificial and allows ELF Relocations
         // ram:00602000-ram:0060202f
         //
            thunk int puts(char * __s)
                Thunked-Function: <EXTERNAL>::puts
    int         EAX:4          <RETURN>
    char *      RDI:8          __s
                puts@@GLIBC_2.2.5
                <EXTERNAL>::puts
```

Complex software often contains more than one binary, including multiple executables and libraries. As such, you may find yourself jumping between various files as you reverse engineer functions that are implemented in one library and called in another library or executable.

Stripped

Sometimes, to save space or to even to obstruct reverse engineering, developers may opt to strip a binary of debugging-related information, including the symbol table. With the Golang compiler, you can do so by passing the `-s` flag (which omits symbol table and debug information) and the `-w` flag (which omits the Debugging With Attributed Record Formats [DWARF] symbol table) to the linker via the `-ldflags` option.

Compile the "Hello World" executable accordingly and note the output when you try to dump the symbols:

```
$ GOARCH=amd64 GOOS=linux go build -ldflags="-s -w" -o stripped hello-world.go
$ objdump -t stripped

stripped:     file format elf64-x86-64

SYMBOL TABLE:
no symbols
```

To see how this affects your reverse engineering process, analyze the binary in the Ghidra CodeBrowser. CodeBrowser jumps to the default entry point that initializes the Go runtime instead of jumping straight to the `main` function because it can no longer reference the symbol for the `main` function. Stripped Golang binaries still include the actual function names in a separate data structure, so it's possible to restore the symbol names using a script; however, this isn't always an option for other programming languages.

For now, you can quickly jump to the `main` function by going to the same virtual address as the one for the `main` function in the unstripped binary. Simply run **`objdump -t hello-world | grep main.main`** to get the virtual address, then use the **`g`** keyboard shortcut in CodeBrowser and go to that address. Other than the function names, both the assembly and the pseudocode should match the unstripped binary.

In short, while stripped binaries can present a significant challenge to reverse engineering, it's still possible to reconstruct the symbol names, either with a script (depending on the compiler) or simply based on what the machine code does. The latter approach requires a good grasp of assembly and the experience to quickly recognize common patterns in standard library functions. Beyond that, you can also look out for logging or error messages that provide more insight into what a particular function does or even contain the name of the function.

Packed

To reduce the size of executables further, developers may sometimes use a packer. *Packers* compress programs into self-contained executables that dynamically unpack, decompress, and execute the original files. The Ultimate

Packer for eXecutables (UPX) is a commonly used packer that you can download from *https://github.com/upx/upx/releases* or install via various package managers.

After downloading UPX, run it on the original Golang "Hello World" binary with **`upx -o hello-world-packed hello-world`**. According to the output, this achieves a rather impressive compression ratio of about 60 percent:

```
        File size         Ratio      Format        Name
   --------------------   ------   -----------   -----------
   1850090 ->   1146320   61.96%   linux/amd64   hello-world-packed

Packed 1 file.
```

However, since the packed binary now runs the UPX decompression routine before executing the actual instructions, it's no longer possible to analyze the original machine code directly.

The most important step in dealing with a packed binary is to first recognize that it has been packed. The next step is to identify which packer was used. In the case of UPX, the initial instructions are well known, and UPX helpfully includes the magic bytes `0x55505821` ("UPX!" in ASCII) in the header. You can observe this in a simple hex dump of the packed binary:

```
> hexdump -C hello-world-packed | head -n 20
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  08 33 5e 00 00 00 00 00  |..>......3^.....|
00000020  40 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  03 00 00 00 00 00 00 00  |....@.8.........|
00000040  01 00 00 00 06 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  00 10 00 00 00 00 00 00  d0 e7 15 00 00 00 00 00  |................|
00000070  00 10 00 00 00 00 00 00  01 00 00 00 05 00 00 00  |................|
00000080  00 00 00 00 00 00 00 00  00 f0 55 00 00 00 00 00  |..........U.....|
00000090  00 f0 55 00 00 00 00 00  e6 4d 08 00 00 00 00 00  |..U......M......|
000000a0  e6 4d 08 00 00 00 00 00  00 10 00 00 00 00 00 00  |.M..............|
000000b0  51 e5 74 64 06 00 00 00  00 00 00 00 00 00 00 00  |Q.td............|
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000000e0  08 00 00 00 00 00 00 00  4f 05 91 f3 55 50 58 21  |........O...UPX!|
000000f0  ec 0a 0e 16 00 00 00 00  ea 3a 1c 00 6a 6c 09 00  |.........:..jl..|
00000100  c8 01 00 00 9d 00 00 00  08 00 00 00 bb fb 20 ff  |.............. .|
00000110  7f 45 4c 46 02 01 01 00  02 00 3e 00 1b 80 ed 45  |.ELF......>....E|
00000120  1f bf 5f da ed 40 2f c8  45 26 38 00 07 0a 17 00  |.._..@/.E&8.....|
00000130  03 3e d8 d7 de 00 06 1e  04 4f 40 00 40 0f 88 01  |.>.......O@.@...|
```

Fortunately, UPX allows you to easily unpack UPX-packed binaries via the `-d` command line option (enter **`upx -d hello-world-packed`** to try it for yourself). Some packers and obfuscators deploy techniques that make it difficult to reverse engineer, such as encrypting data with randomized values, and may require you to dump the unpacked and decrypted bytes from memory or analyze the unpacking routine in detail. While you'll encounter this more commonly with malware, it helps to be prepared to recognize a situation where a packer has been used and to know how to approach such binaries.

Conclusion

In this chapter, you navigated a wide range of binaries across different categories, including scripts, intermediate representations, and machine code. You also reverse engineered simple examples of each type with appropriate tools and techniques.

As you target larger and more complex software, like firmware, you may need to juggle multiple types of binaries. For example, an Android application written in JavaScript (React Native) may call a native module compiled from Java, which in turn can call C++ libraries via the Java Native Interface. This is why it's essential to build breadth rather than focusing too much on techniques that may apply to only a small subset of binaries.

While this was hardly an exhaustive introduction to all the types of binaries you'll encounter, you should be able to generalize some of the approaches used regardless of the programming language or compilation. For example, keep an eye out for sources of metadata that can help you analyze the machine code more effectively or even decompile it to source code. Pay attention to language- or compiler-specific quirks and optimizations that can assist you in identifying function names and imports. Look for clues that a developer has used a packer or obfuscator, identify the tool used, and read the documentation to find out how to reverse it, if possible. These tips will help you identify the most important or useful parts of the program to reverse engineer first, which is a topic the next chapter will explore in greater detail.