Protostar Introducing memory corruption in Linux/x86

About

Protostar introduces the following in a friendly way:

- Network programming
  - Byte order
  - Handling sockets
- Stack overflows
- Format strings
- Heap overflows

The above is introduced in a simple way, starting with simple memory corruption and modification, function redirection, and finally executing custom shellcode.

In order to make this as easy as possible to introduce Address Space Layout Randomisation and Non-Executable memory has been disabled. If you are interested in covering ASLR and NX memory, please see the Fusion page.

Download

Downloads are available from the download page

Getting started

Once the virtual machine has booted, you are able to log in as the "user" account with the password "user" (without the quotes).

The levels to be exploited can be found in the /opt/protostar/bin directory.

For debugging the final levels, you can log in as root with password "godmode" (without the quotes)

Core Files

README! The /proc/sys/kernel/core_pattern is set to /tmp/core.%s.%e.%p. This means that instead of the general ./core file you get, it will be in a different directory and different file name.

Protostar stack0

About

This level introduces the concept that memory can be accessed outside of its allocated region, how the stack variables are laid out, and that modifying outside of the allocated memory can modify program execution.

This level is at /opt/protostar/bin/stack0

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer);

  if(modified != 0) {
    printf("you have changed the 'modified' variable\n");
  } else {
    printf("Try again?\n");
  }
}
```

Discussion

Protostar stack1

About

This level looks at the concept of modifying variables to specific values in the program, and how the variables are laid out in memory.

Hints:

- If you are unfamiliar with the hexadecimal being displayed, "man ascii" is your friend.
- Protostar is little endian

This level is at /opt/protostar/bin/stack1

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  if(argc == 1) {
    errx(1, "please specify an argument\n");
  }

  modified = 0;
  strcpy(buffer, argv[1]);

  if(modified == 0x61626364) {
    printf("you have correctly got the variable to the right value\n");
  } else {
    printf("Try again, you got 0x%08x\n", modified);
  }
}
```

Discussion

Protostar stack2

About

Stack2 looks at environment variables, and how they can be set.

This level is at /opt/protostar/bin/stack2

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];
  char *variable;

  variable = getenv("GREENIE");

  if(variable == NULL) {
    errx(1, "please set the GREENIE environment variable\n");
  }

  modified = 0;

  strcpy(buffer, variable);

  if(modified == 0x0d0a0d0a) {
    printf("you have correctly modified the variable\n");
  } else {
    printf("Try again, you got 0x%08x\n", modified);
  }

}
```

Discussion

Protostar stack3

About

Stack3 looks at environment variables, and how they can be set, and overwriting function pointers stored on the stack (as a prelude to overwriting the saved EIP)

Hints:

- both gdb and objdump is your friend you determining where the win() function lies in memory.

This level is at /opt/protostar/bin/stack3

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
    printf("calling function pointer, jumping to 0x%08x\n", fp);
    fp();
  }
}
```

Discussion

Protostar stack4

About

Stack4 takes a look at overwriting saved EIP and standard buffer overflows.

Hints:

- A variety of introductory papers into buffer overflows may help.
- gdb lets you do "run < input"
- EIP is not directly after the end of buffer, compiler padding can also increase the size.

This level is at /opt/protostar/bin/stack4

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

Discussion

Protostar stack5

About

Stack5 is a standard buffer overflow, this time introducing shellcode.

Hints:

- At this point in time, it might be easier to use someone elses shellcode
- If debugging the shellcode, use \xcc (int3) to stop the program executing and return to the debugger
- remove the int3s once your shellcode is done.

This level is at /opt/protostar/bin/stack5

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

Discussion

Protostar stack6

About

Stack6 looks at what happens when you have restrictions on the return address.

This level can be done in a couple of ways, such as finding the duplicate of the payload (*objdump -s* will help with this), or *ret2libc* , or even return orientated programming.

It is strongly suggested you experiment with multiple ways of getting your code to execute here.

This level is at /opt/protostar/bin/stack6

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void getpath()
{
  char buffer[64];
  unsigned int ret;

  printf("input path please: "); fflush(stdout);

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xbf000000) == 0xbf000000) {
    printf("bzzzt (%p)\n", ret);
    _exit(1);
  }

  printf("got path %s\n", buffer);
}

int main(int argc, char **argv)
{
  getpath();



}
```

Discussion

Protostar stack7

About

Stack6 introduces return to .text to gain code execution.

The metasploit tool "msfelfscan" can make searching for suitable instructions very easy, otherwise looking through objdump output will suffice.

This level is at /opt/protostar/bin/stack7

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

char *getpath()
{
  char buffer[64];
  unsigned int ret;

  printf("input path please: "); fflush(stdout);

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xb0000000) == 0xb0000000) {
    printf("bzzzt (%p)\n", ret);
    _exit(1);
  }

  printf("got path %s\n", buffer);
  return strdup(buffer);
}

int main(int argc, char **argv)
{
  getpath();



}
```

Discussion

THIS PAGE LEFT BLANK INTENTIONALLY

THIS PAGE LEFT BLANK INTENTIONALLY

Protostar format0

About

This level introduces format strings, and how attacker supplied format strings can modify the execution flow of programs.

Hints:

- This level should be done in less than 10 bytes of input.
- "Exploiting format string vulnerabilities"

This level is at /opt/protostar/bin/format0

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void vuln(char *string)
{
  volatile int target;
  char buffer[64];

  target = 0;

  sprintf(buffer, string);

  if(target == 0xdeadbeef) {
    printf("you have hit the target correctly :)\n");
  }
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

Discussion

Protostar format1

About

This level shows how format strings can be used to modify arbitrary memory locations.

Hints: objdump -t is your friend, and your input string lies far up the stack :)

This level is at /opt/protostar/bin/format1

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void vuln(char *string)
{
  printf(string);

  if(target) {
    printf("you have modified the target :)\n");
  }
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

Discussion

Protostar format2

About

This level moves on from format1 and shows how specific values can be written in memory.

This level is at /opt/protostar/bin/format2

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);
  printf(buffer);

  if(target == 64) {
    printf("you have modified the target :)\n");
  } else {
    printf("target is %d :(\n", target);
  }
}

int main(int argc, char **argv)
{
  vuln();
}
```

Discussion

Protostar format3

About

This level advances from format2 and shows how to write more than 1 or 2 bytes of memory to the process. This also teaches you to carefully control what data is being written to the process memory.

This level is at /opt/protostar/bin/format3

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void printbuffer(char *string)
{
  printf(string);
}

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);

  printbuffer(buffer);

  if(target == 0x01025544) {
    printf("you have modified the target :)\n");
  } else {
    printf("target is %08x :(\n", target);
  }
}

int main(int argc, char **argv)
{
  vuln();
}
```

Discussion

Protostar format4

About

format4 looks at one method of redirecting execution in a process.

Hints: objdump -TR is your friend

This level is at /opt/protostar/bin/format4

Source code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void hello()
{
  printf("code execution redirected! you win\n");
  _exit(1);
}

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);

  printf(buffer);

  exit(1);
}

int main(int argc, char **argv)
{
  vuln();
}
```

Discussion

THIS PAGE LEFT BLANK INTENTIONALLY

THIS PAGE LEFT BLANK INTENTIONALLY

Protostar heap0

About

This level introduces heap overflows and how they can influence code flow.

This level is at /opt/protostar/bin/heap0

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct data {
  char name[64];
};

struct fp {
  int (*fp)();
};

void winner()
{
  printf("level passed\n");
}

void nowinner()
{
  printf("level has not been passed\n");
}

int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;

  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;

  printf("data is at %p, fp is at %p\n", d, f);

  strcpy(d->name, argv[1]);

  f->fp();

}

```

Discussion

Protostar heap1

About

This level takes a look at code flow hijacking in data overwrite cases.

This level is at /opt/protostar/bin/heap1

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>



struct internet {
  int priority;
  char *name;
};

void winner()
{
  printf("and we have a winner @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  struct internet *i1, *i2, *i3;

  i1 = malloc(sizeof(struct internet));
  i1->priority = 1;
  i1->name = malloc(8);

  i2 = malloc(sizeof(struct internet));
  i2->priority = 2;
  i2->name = malloc(8);

  strcpy(i1->name, argv[1]);
  strcpy(i2->name, argv[2]);

  printf("and that's a wrap folks!\n");
}

```

Discussion

Protostar heap2

About

This level examines what can happen when heap pointers are stale.

This level is completed when you see the "you have logged in already!" message

This level is at /opt/protostar/bin/heap2

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
    printf("[ auth = %p, service = %p ]\n", auth, service);

    if(fgets(line, sizeof(line), stdin) == NULL) break;

    if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
    }
    if(strncmp(line, "reset", 5) == 0) {
      free(auth);
    }
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if(strncmp(line, "login", 5) == 0) {
      if(auth->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}

```

Discussion

Protostar heap3

About

This level introduces the Doug Lea Malloc (dlmalloc) and how heap meta data can be modified to change program execution.

This level is at /opt/protostar/bin/heap3

Source code

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
  printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  char *a, *b, *c;

  a = malloc(32);
  b = malloc(32);
  c = malloc(32);

  strcpy(a, argv[1]);
  strcpy(b, argv[2]);
  strcpy(c, argv[3]);

  free(c);
  free(b);
  free(a);

  printf("dynamite failed?\n");
}

```

Discussion