***From Buffer Overflow to JIT-Spray-ROP (JIT Code Reuse with JIT-Sprayed gadgets)***

***Daniel Bloom***

# Chapter 0: Abstract

Buffer Overflow attacks are a type of attack that stems from invalid/improper processing of user-controlled variables/input fields. These types of attacks occur when developers don't validate whether or not the inputted data is the same size (or less) as the required amount. For example, if a program takes user input into a char[64] variable but the program *doesn't validate* whether the length of the inputted data is less than or equal to 64 (*or the program uses an insecure user-interaction method such as gets() in C*), the user could continue to input data beyond the 64 character limit and effectively overwrite arbitrary data in the field of memory the input variable resides in (usually the stack). This can lead to all types of attacks – from basic denial of service, to malicious code execution, to more advanced attacks like Return-to-libc (Ret2Libc), etc. This project will detail the methodology and theory behind one of the most complex attacks that can stem from Buffer Overflow vulnerabilities, JIT-Spray-ROP (Return Oriented Programming on Just-In-Time Compiled code (also known as JIT Code Reuse) where gadgets are custom generated through JIT-Spraying) attacks.

The JIT-Spray-ROP attack works by utilizing JIT compiled custom Return Oriented Programming (ROP) gadgets generated through a JIT-Spraying attack to create custom ROP gadget chains during runtime. Then, by utilizing a buffer overflow vulnerability, a malicious agent can inject the ROP chain and execute the newly-built malicious code, *bypassing both DEP and ASLR*. In traditional ROP attacks, an attacker may analyze an offline/inactive binary (static analysis) in order to identify usable ROP gadgets for exploit development. However, in JIT-Spray-ROP attacks the ROP gadgets the malicious agent wants to use are pre-defined by the malicious agent and then generated within the victim application *during runtime*. This means that through this attack, malicious agents are able to inject and execute custom malicious code without actually injecting a malicious script or function, all while bypassing ASLR and DEP. Furthermore, while implementations of JIT-ROP attacks that do not include JIT-Spraying require a memory disclosure vulnerability (such as a printf vulnerability) to leak large amounts of memory pages which are then put through a separate process that searches through them to find

usable ROP gadgets, the JIT-Spray-ROP attack bypasses the need for a memory disclosure vulnerability altogether by utilizing a JIT-Spray attack to fill a large area of memory with ROP gadgets and then redirecting code flow to a ROP chain of JIT-Sprayed gadgets.

In addition to the overall explanation of the JIT-Spray-ROP attack and all the pre-requisite information described throughout this paper, a practical example is provided which utilizes this attack to exploit CVE-2012-4787, a vulnerability found in Windows 7's IE9/10

**A Note on Formatting:** This article is formatted as the story of "Ray Skillman", a 31337 h4x0r that has been tasked with an espionage mission in which he has to steal the best available robot-manufacturing machine and identify its weakness. Each section (up until the discussion of CVE-2012-4787) is a "chapter" of the story that will start out with Ray Skillman's adventure before then continuing on to a more technical explanation. This is to make complex concepts (such as JIT compilation, Return oriented programming, etc.) more easily understandable. Furthermore, considering this technique is fairly advanced, I tried to add as much pre-requisite information as possible to give the reader enough background information to support a full understanding of the subject matter.

# Ray Skillman Chapter 1: Getting Ready

*Ray Skillman is the world's most infamous hacker. Every day as he gears up for some ultra-hacking adventures he finds himself needing to take some things with him – after all, no self-respecting hacker would leave his hacking tools behind! Small things, like his rubber ducky or his balaclava, can go in the pockets of his leather trench coat, while larger things - like his elite encrypted laptop with all the latest hacking tools bookmarked in his tor browser - need to go in his backpack (a black Jansport). Now that he's geared up, all Ray Skillman needs to do is put his drive-by hacking machine in the trunk of his Chevy Spark and head out for his espionage mission.*


Much like the above scenario, when computer programs are on-the-go (processing data, creating and manipulating variables, etc.) they need places to store things! Unfortunately, computers aren't able to store variables, arrays, and other data elements in cars, trench coats, or Jansports – instead, they use a number of previously laid-out fields of memory called the **registers**, **the Stack**, and **the Heap**. The registers, which are smaller than either the Stack or the Heap, can store up to 32 bits of data (for x86 systems) and can be seen as the computer's very own trench coat pockets (used for storing smaller pieces of data that the computer needs quick/immediate access to). On the other hand, the Stack, which is a region of memory that the program uses to store most variables and other data objects, can hold up to 1 megabyte on Windows and up to 8 megabytes on linux, and can be thought of as the computer's very own black Jansport. Furthermore, the stack is set up in a "Last-In-First-Out" fashion, meaning the last thing "pushed" (added to) the stack will be the first thing "popped" (removed) from it. You can think of this like a dishwasher washing a stack of dishes, the washer will always wash the last plate added to the stack (the one on the top) first and so on (Rather than taking a plate from the bottom or middle of the stack). Lastly, the Heap is a CPU-managed region of memory that the application may allocate space within if it needs to store larger pieces of data and can be thought of as the Computer's very own Chevy Spark.

# Ray Skillman Chapter 2: The Encounter

*After driving far too fast in residential areas, Ray Skillman has finally made it to the secret hideout his target – Big Richard Fauquier, the infamous cyber-Mafioso from Fauquier County, VA – is using. Ray hops out of his car and grabs his EMP gun (Ray kills computers, not people) and his Jansport as he runs into the hideout! His mission is to stop a robot manufacturing sale and steal the best,*

*most efficient, robot manufacturing machine so that he can discover its weakness. As he sneaks into the hideout he finds two hallways each leading to a different robot manufacturing sale - clearly Big Ricky was shopping around for the best deal! Ray quickly and quietly sneaks down the left hallway and eavesdrops:*

*Big Ricky's representative – "Alright, show me what you got"*

*Salesman – "Alright, this is the Comm-Piler3000, it'll take your human speech and turn it into whatever robot you describe! To use it, write down everything you would want in a robot then read it out-loud while holding down this big green button."*

**The salesman reads a robot description from his notebook and the machine creates the robot**

*Ray quickly turns around and runs over to the sale going on at the end of the right hallway. As he creeps around the corner he begins to hear the sale:*

*Big Ricky's representative – "Sorry we didn't start on time; I could have sworn I saw somebody but I guess it was nothing. Go on, show me what you got!"*

*Salesman – "Alright, this is the JIT-Piler9999 (That's right, 9999, those other "stupidname3000" hacks have nothing on us!), basically how it works is that you describe a robot and as you describe it, in real time, the machine will build that robot. That means you can come in with a half formed idea and then try different options along the way. Also, this is far better suited for on-the-fly robot creation than the slow, full-description-necessary, robot makers are competitors might try to sell you!"*

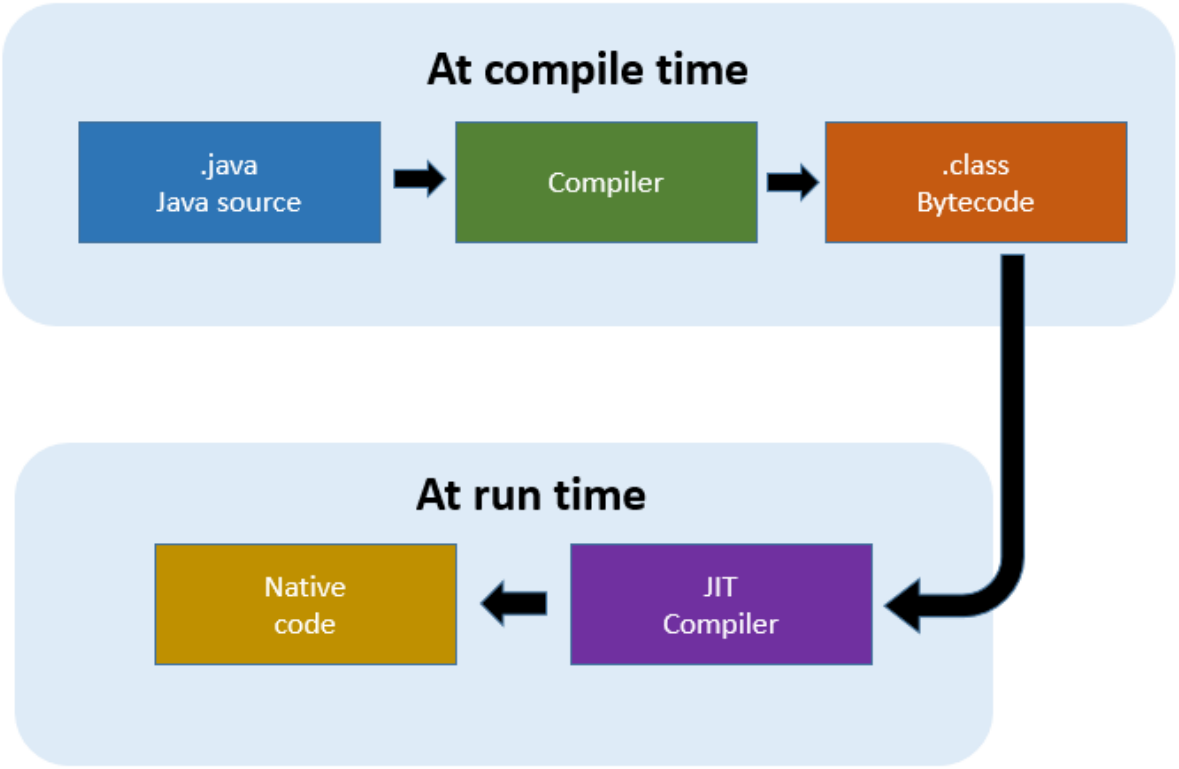
**Big Ricky's representative begins to describe a totally random robot with all kinds of weird and wacky features and the robot maker builds the robot.*

While Ray Skillman's wild robot-sales espionage adventure seems farfetched to most, it actually has many parallels to the very real, and equally wild, world of **standard code compilation** and **Just in Time (JIT) Compilation**. A standard code compiler, much like the Comm-Piler3000, takes pre-written code (i.e. c code) and translates it into machine code. In standard compilation, code cannot be written and compiled at the same time, it must be pre-written and then put through the compilation process. In JIT compilation, on the other hand, code is compiled during runtime. This runtime compilation, which usually takes place within a virtualized environment, gives the program a

substantial performance boost but also adds an attack surface that, if exploited, could lead to the execution of custom (malicious) native code.

While in some cases JIT compilation can be done on-the-fly through the use of JIT-enabled primitives (i.e. “var a = (0x11223344^0x44332211)” may be automatically compiled into “mov 0x11223344, eax; xor 0x44332211, eax;”), JIT compilation normally takes place as a means of execution optimization (i.e. if a piece of code is run over and over, it may be interpreted at first but after a while the JIT compiler will convert it into native code for quicker execution). While JIT compilation is prevalent in many popular programming languages / platforms, the most common/easy to explain implementation would be the JIT compiler implemented into the Java Runtime Environment.

When Java code is compiled, it is turned into a .class file, which are platform-agnostic files that contain the bytecode of the compiled .java file. This bytecode can then be interpreted by the JVM (Java’s virtualized environment), which loads these class files and executes the bytecode. During execution, the JIT compiler may improve the overall performance of the java program by compiling pieces of bytecode into native machine code.



Once a piece of java code has been JIT compiled, the JVM will execute the native machine code, rather than interpret the bytecode found within the .class files.

# Ray Skillman Chapter 3: Dead-Piler3000

*Now that Ray has eavesdropped on both sales he's realized that the JIT-Piler9999 is clearly the best of the two robot makers. Before stealing it, Ray needs to get rid of the salesman and Big Ricky's men! Ray quickly fires his EMP gun into the ceiling, shutting off all the lights! Luckily, Ray has spent so much time on the dark web that he developed night-vision!*

*Boom! Pow! Ouch! Oof!*

*Ray takes down all 4 people in the room right as the EMP bullet wares off and the lights come back on. Breathing heavily, Ray looks toward the Jit-Piler9999 excited to discover its weakness! However, as Ray begins to step towards I, a body comes flying past him! Ray turns to find the robot that the Comm-Piler3000's salesman had built looking particularly angry. Luckily, Ray is a super hacker and knows exactly what to do! While Ray was eavesdropping he realized that the salesman had added a feature that allowed anything to be loaded into the robot's arm and fired out of it. All Ray had to do was clog the robot's arm with data-bombs (bombs that shoot out data and make robots execute the data if they get caught in the bomb's radius) so that it couldn't fire and just wait for them all to go off!*

*Ray did some intense parkour and landed on the robots back! He quickly began to shove data bombs into the robot's arm until they got stuck and the robot couldn't fire them out, then he back flipped off the robot and clicked the detonate button!*

*The robot stood for a second completely motionless before violently shaking, and then it froze again. Ray had used a new payload in his data bombs this time, one that would use the robot's own code against it by taking over its body and making it destroy itself!*

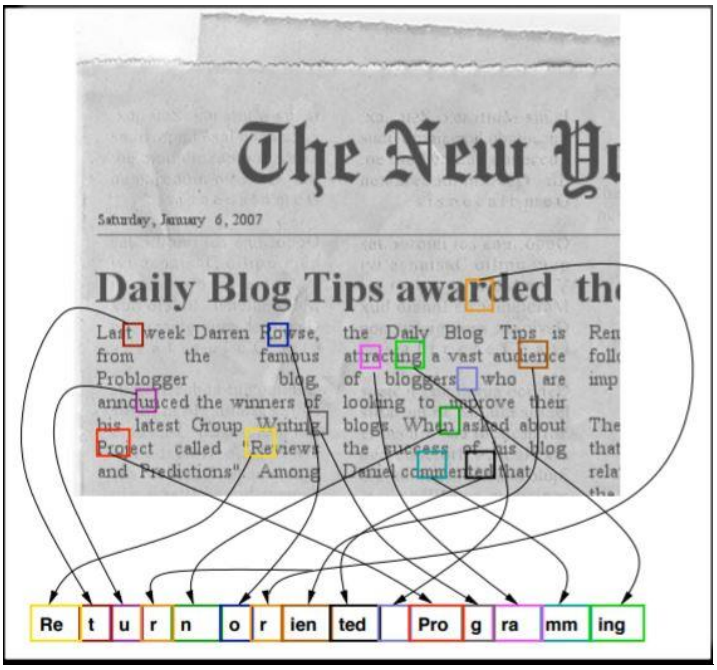
*Once the robot was defeated, Ray quickly turned around as he had forgotten about the robot created by the JIT-Piler9999! However, to his great luck, the robot was simply a robot-Peter Dinklage with Gilbert Gottfried's voice. Clearly he was angry but Ray just laughed,*

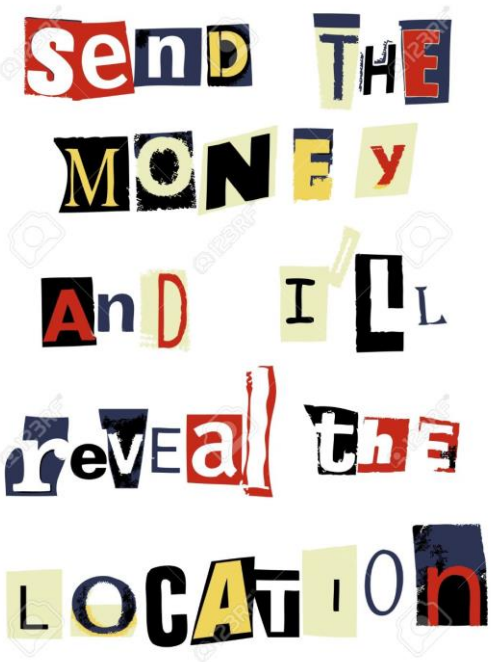
*packed the JIT-Piler9999 into his Chevy Spark and drove back to his base.*

Wow! Ray sure is a cool hacker! But you know what's even cooler? The parallels we can draw between Ray's adventure and **Buffer Overflow** / **Flow-Hijacking** attacks!

Buffer Overflow vulnerabilities are present when programs don't sanitize or check the data being inputted into certain fields. Much like the Comm-Piler3000's robot, if a program doesn't check what's entering a field (i.e. a variable, buffer (a region of memory set specifically for certain data pieces (think of it like an array)), or in the case of the Comm-Piler3000's robot – an arm), a malicious agent can overflow (add more data than expected / can be processed) the field and begin writing over arbitrary data. This can cause the program to act in a number of different ways depending on the payload the malicious agent chooses. Some buffer overflow attacks can simply lead to the program crashing as it fails with an out-of-bounds error, while other, more sophisticated, types of Buffer overflow attacks can actually manipulate the program's execution flow and execute custom malicious code. This type of control flow hijacking can allow malicious agents to execute code on the stack, in the heap, or – as in the case of the Comm-Piler3000's robot – can even lead to more sophisticated that hijack the control-flow of the program and force it to act in different, unintended, and often malicious ways (in Ray's adventure this can be seen when the Data bombs hijack the robot's 'mind' and force it to destroy itself).

While control-flow hijacking can take many shapes (i.e. inject and redirect to custom shellcode, return to libc attacks, etc.), the main control-flow hijacking attack that we will be covering in this article is the Return Oriented Programming attack.




Return Oriented Programming is a control-flow hijacking attack that involves re-using benign code already found within the victim application to create a chain of small code snippets (called Gadgets) that, once executed in order, constitute a fully formed, "independent program" (not really independent as it relies on the otherwise benign code found within the victim application). Malicious agents may use ROP attacks to force a program to execute malicious code without actually injecting any malicious code into the program. One way of thinking of ROP attacks could be like ransom letters that have individual letters/words cut and pasted from different sources (magazines, newspapers, etc.) to create the letter.

To execute Return Oriented Programming attacks, malicious agents typically perform some static analysis on the victim binary in order to identify ROP gadgets that would assist in the execution of malicious activity. Once located, the malicious agent will create a chain of these gadgets and execute a control-flow hijacking attack (through means of a buffer overflow or other vulnerability) that will force the program to execute each gadget on the chain in order. The end result of this process is custom malicious code execution without actually injecting any code.

# Ray Skillman Chapter 4: Risk Analysis

*Once home, Ray immediately pulls out the JIT-Piler9999 and begins to try to analyze its attack surface so he can begin looking for vulnerabilities. Ray's ultimate goal is to find an attack that will prevent the JIT-Piler9999 from ever creating a functional robot ever again! While reading the JIT-Piler9999's user-manual, he notices that he can fine-tune the JIT-Piler9999's Robot-Part Functionality Mechanism (The part that programs how each of the robot's parts should function) so that it would only process pieces of the user's command rather than the entire command. To test this, Ray sets up the machine to only process the user's command after the fifth word in the sentence and then says "a robot that is not unstable and will destroy itself".*

*Wirrr…ZZZZ…BUZZZZZZZZZZ!*

*An alarm goes off and the JIT-Piler9999 spits out a red piece of paper. 'This isn't a robot…', Ray thinks to himself as he picks up the paper.*

*[ERROR] The JIT-Piler9999 does not permit the creation of self-destructing robots.*

*Ray, looking puzzled, thinks to himself for a second before saying "a robot that is not too complex and will clog the JIT-Piler9999's output gate for 10 seconds before shrinking."*

*Wirrr…ZZZZ…Ding!*

*The JIT-Piler9999 begins to assemble the most complex robot Ray has ever seen. As it finishes assembling all of the pieces, the JIT-Piler9999 tries to push the robot through the output gate but it simply won't fit, the robot is stuck in the assembly unit! However, after 10 seconds the robot quickly shrinks down and is able to pass through the gate with ease.*

*Upon seeing this, Ray has a fantastic idea and excitedly says "a robot that is not too complex and will clog the JIT-Piler9999's output gate while also executing the payload found within this data bomb"*

*A compartment on the side of the JIT-Piler9999 immediately pops open and Ray puts his data bomb in it. The JIT-Piler9999 scans the bomb before beginning to assemble the robot.*

*Wirrr…ZZZZ…Ding!... BOOM*

*As the JIT-Piler9999 finishes up the assembly of the robot, it plugs itself into one of the robot's ports and uploads the data bomb*

*payload. Then, as the robot begins to power on, the JIT-Piler9999 unsuccessfully attempts to push it through the output gate. The robot begins to flail and thrash as the data bomb's payload executes an it tries to destroy itself. After a couple loud bangs the entire JIT-Piler9999 explodes!*

*"Hacktastic!" yells Ray, he's finally destroyed the JIT-Piler9999!*

Ray has finally completed his mission and created an attack that will successfully destroy the JIT-Piler9999! While, unfortunately, we don't have such things as data bombs or on-the-fly robot creators, there is an attack that very closely resembles Ray's wild antics – the JIT-Spray-ROP attack. This attack combines **JIT Spraying** and **Return Oriented Programming** to successfully bypass ASLR and DEP.

JIT Spraying is an attack much like Ray's 'only process after word X' attack in which malicious agents fill JIT enabled primitives with hex values that correspond with assembly instructions and then redirect code flow into one of the variables at an offset that forces the program to process the data as said assembly instructions rather than as variable data.

For example, in Javascript's ASM.JS, the code:

```
X = (X + 0xA8909090)|0;
X = (X + 0xA8909090)|0;
```

Is JIT compiled into:

```
00: 0590909 0A8   ADD EAX, 0Xa8909090
05: 0590909 0A8   ADD EAX, 0Xa8909090
```

While this instruction may initially seem completely benign, if a malicious agent were to hijack the program's control flow and redirect into offset 01, the instruction is then processed as:

```
01: 90     NOP
02: 90     NOP
03: 90     NOP
04: A805   TEST AL, 05
06: 90     NOP
07: 90     NOP
08: 90     NOP
09: A8…
```

As we can see, much like Ray's malicious command, if we jump into the middle of the instruction rather than at the beginning, the program will process the data completely differently.

While in the past JIT-Spraying alone would be enough of an exploit to take down most applications, modern defenses render (such as fine-grained ASLR, random NOP placements that break long shellcode, etc.) JIT-Spraying attacks unable to reliably generate executable code. However, this problem can be bypassed through the additional use of Return Oriented Programming.

As we discussed earlier, Return Oriented Programming is an attack that utilizes gadgets (small snippets of code) already found within the victim application to create a chain of commands that, when executed in order, constitute an "independent program". To utilize a ROP chain to bypass DEP/other security mechanisms when executing a JIT-Spray vulnerability a malicious agent, rather than JIT-Spraying custom shellcode that would then be executed, JIT-Sprays a large amount of small ROP gadgets. These gadgets, which are small enough to not have a random NOP statement placed in them by the JIT compiler, are then used to create a full ROP chain. Once the chain is built, the malicious agent simply hijacks the victim application's control flow through the use of some other vulnerability (i.e. buffer overflow) and redirects to an offset within the JIT-Sprayed field that would force the application to process the data as assembly instructions.

# Ray Skillman Chapter 5: Unhappy CEO

*Upon learning that Ray Skillman – one of the most notorious super-hackers on the planet – has managed to find a way to make the JIT-Piler9999 destroy itself, Hank Business – the CEO of JIT Corp (the creators of the JIT-Piler9999) – is determined to find a way to defend against the attack.*

*After many fruitless attempts at different defensive mechanism, Hank Business decided that the best way to prevent Ray's attack from being effective would be to:*

- *force the JIT-Piler9999 to always process commands from the beginning of the sentence*
- *Prevent commands that would get the robot stuck inside the JIT-Piler9999*

- *Reconfigure the robot's control system so that Ray's data bomb wasn't able to use the robot's own functions against it.*

*While these additions make the JIT-Piler9999 slightly slower, Hank Businesses is just happy Ray's attack no longer works!*

I know, I know – This section has a far weaker Ray Skillman example than the rest. However, the defenses Hank Business decided to use against Rays attack do seem pretty similar to modern defenses aiming at preventing JIT-ROP and JIT-Spray-ROP attacks in general!

While there aren't many reliable ways of preventing ROP attacks in general (as the code being used to exploit the malicious functions is benign, often signed, pre-existing application code), and much more-so when JIT compilation is involved, there are a number of improvements to ASLR implementations that make identifying ROP gadgets extremely difficult to identify or generate. One framework in particular, called "Shuffler" would be particularly beneficial in defending against JIT-Spray-ROP vulnerabilities. Shuffler, which was released in 2016, aims to take fine-grained ASLR (randomization of instructions as well as whole functions) and randomizing address locations every 20-50 milliseconds (rather than randomizing addresses only upon program initialization). This would significantly reduce the attack surface of applications that would otherwise be vulnerable to a JIT-Spray-ROP attacks by making it impossible to predict where a gadget might be even after generating an initial ROP chain. Furthermore, for JIT-ROP attacks that do not utilize custom/injected ROP gadgets, defenses such as Instruction Re-Ordering/Location Randomization, Instruction Substitution, and Register Re-Allocation all make the malicious agent's job far more difficult, as the amount of usable gadgets found with a victim application would be severely lowered (as Instruction Substitution and Register Re-Allocation would reform once-valid ROP gadgets into unusable ones) and their locations would be unpredictable.

# Practical Example: CVE-2012-4787

The JIT-Spray-ROP attack can be executed to exploit CVE-2012-4787, a vulnerability that affects IE 9 on Windows 7. This attack, which was detailed in Fermin J. Serna's 2013 paper, "Flash JIT – Spraying info leak gadgets."

CVE-2012-4787 is a "Use-After-Free" vulnerability in Microsoft's Internet Explorer 9 and 10 which allows remote attackers to execute

arbitrary code on a victim's system by having them navigate to a malicious webpage. This vulnerability stems from the fact that IE9/10 failed to properly process an object that had been incorrectly initialized or deleted when said object is added to the style attributes array. This allows a remote malicious agent to execute arbitrary code on the victim's system.

This attack is performed by first JIT-spraying a large number of ROP gadget and then redirecting the program's control flow to execute a ROP-chain generated by the JIT-compiled gadgets.

In our demo, we used the proof of concept code that accompanied Serna's 2013 paper (http://zhodiac.hispahack.com/my-stuff/security/Flash_Jit_InfoLeak_Gadgets). The ROP chain generated by this PoC performs the following actions in order to leak information from the victim application (*please note: while this exploit seeks only to leak application information, this attack can be used to execute arbitrary code on the victim's system, allowing a remote attacker to completely hijack the machine):*

- Pop an address from the stack (return address once the JITed code gets executed)
- Push it not to alter the flow of execution
- Store it inside a heap spray (at a fixed address or relative to a register)
- Clean up the program so it doesn't crash
- Return the flow of execution to the attacked program

References:


- https://www.greencarreports.com/news/1084953_first-2014-chevy-spark-ev-sold-why-we-chose-this-electric-car
- https://medium.com/@danielabloom/binary-exploitation-eli5-part-3-d1872eef71b3
- https://medium.com/@danielabloom/binary-exploitation-eli5-part-2-8fd71bf214b9
- https://hackernoon.com/binary-exploitation-eli5-part-1-9bc23855a3d8
- https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/
- https://media.blackhat.com/us-13/US-13-Snow-Just-In-Time-Code-Reuse-Slides.pdf
- http://zhodiac.hispahack.com/my-stuff/security/Flash_Jit_InfoLeak_Gadgets.pdf
- https://rh0dev.github.io/blog/2017/the-return-of-the-jit/
- https://rh0dev.github.io/blog/2017/the-return-of-the-jit-part-2/
- https://www.exploit-db.com/exploits/44293
- http://www.cs.columbia.edu/~junfeng/papers/shuffler-osdi16.pdf
- https://nvd.nist.gov/vuln/detail/CVE-2012-4787