Heinrich C. Mayr, Martin Pinzger (Hrsg.): INFORMATIK 2016,
Lecture Notes in Informatics (LNI), Gesellschaft fr Informatik, Bonn 2016 2077

Sigreturn Oriented Programming is a real Threat

Rémi Mabon[1]

Abstract: This paper shows that Sigreturn Oriented Programming (SROP), which consists of using calls to *sigreturn* to execute arbitrary code, is a powerful method for the development of exploits. This is demonstrated by developing two different kinds of SROP based exploits, one asterisk exploit which was already portrayed in the paper presenting SROP, and one novel exploit for a recently disclosed bug in the DNS address resolution of the default GNU C library. Taking advantage of the fact, that these exploits have very few dependencies on the program being exploited, a library is implemented to automate wide parts of SROP exploit creation. This highlights the potential of SROP in respect to reusable and portable exploit code which strongly supports the conclusion of the original paper: SROP is a real threat!

1 Introduction

In the world of security, there is a constant cat and mouse game between programmers, which want to make software safer and attackers, which try to exploit vulnerabilities. This paper focuses on the Sigreturn Oriented Programming (SROP) technique presented by Erik Bosman and Herbert Bos in 2014 [BB14].
First, the different preconditions and ways to exploit SROP are shown in section 2. To demonstrate the use of both SROP exploitation techniques on real targets, section 3 focuses on the implementation of the SROP Chain based Asterisk exploit mentioned in [BB14] and the following section 4 demonstrates the use of an SROP based attack using a known address on a target which resolves DNS addresses with glibc. The reusability of SROP attacks is then demonstrated in section 5 by analyzing how much of the code used in the previous exploits can be reused to exploit a self made exploitable binary. This results in a reusable library for SROP attacks. Following this, section 6 gives an estimate on how much some defence mechanisms mitigate the danger of SROP attacks. Finally, a conclusion is given in section 7.

2 Sigreturn Oriented Programming

Sigreturn calls exist in UNIX to restore the user context after a program is interrupted by a signal. The signal handling procedure as seen by the program looks as follows [BB14]:

[1] Furtwangen University, Robert-Gerwig-Platz 1, 78120 Furtwangen im Schwarzwald, remi.mabon@hs-furtwangen.de

- A signal frame is added to the stack. This frame contains the current value of all registers.
- A new return address is added to the top of the stack. This return address simply leads to a small stub which executes the *sigreturn* system call.
- The signal handler is called. What the signal handler does depends on the signal received.
- After the signal handler is done, the return address is used to execute the *sigreturn* system call if the program wasn't terminated.
- The *sigreturn* call uses the signal frame to restore all registers to their previous state.
- Finally, the execution of the program is continued exactly as before.

SROP exploits the fact, that UNIX systems don't check if a signal was sent when a program executes the *sigreturn* system call. Since the *sigreturn* call restores all registers from stack, this can be extremely powerful when the attacker has control of the stack as all registers can be set to attacker controlled value with one single gadget: *syscall* and *return*. Furthermore, this gadget can always be found at the same locations on some systems, e.g. Linux with a kernel older than 3.3. A list of systems where this is the case is provided in [BB14].The attack is further facilitated by the fact that the *sigreturn* call is executed by using the processor instruction *syscall* which can also execute all other system calls depending only on the value of the *AX/EAX/RAX* register (the name of the register varies depending on the architecture). This way, by chaining *sigreturn* and other system calls, arbitrary code can be executed with only this one gadget. As demonstrated in [BB14], such a chain is even Turing complete without using any shellcode.
Depending on the concrete situation, there are at least two possible sets of preconditions needed to execute arbitrary code using *sigreturn* . The first one can be used if the attacker knows an address with user controlled data and is shown in subsection 2.1. The second one bypasses this need by using an SROP chain but requires a known file descriptor to which the attacker can send data. This is shown in subsection 2.2.

2.1 Known address

To execute arbitrary code using a known address with user controlled content, the following preconditions have to be met [BB14]:

1. The attacker needs a way to control the instruction pointer, e.g. through a *return* instruction on stack overwritten by the attacker.
2. The stack pointer has to be located on data controlled by the attacker where adding *NULL* bytes is possible.
3. The attacker has to know the address of a piece of data he controls. This can either be on the stack or in any other user controlled location e.g. some buffer.

4. Either the address of a *sigreturn* gadget or the address of a *syscall* gadget and a way to set the *AX/EAX/RAX* to the right value have to be known.

The preconditions 1 and 2 can be met with a simple stack overflow, provided that the vulnerable code allows *NULL* bytes. The precondition 3 can be met easily in most cases if the exact version of the program is known and the application is not compiled as position independent executable, since the location of buffers and other data pieces can be looked up by debugging the exact same version of the program locally. If this is not possible or the program was compiled using the *PIE* flag, a memory disclosure vulnerability may be needed to get a valid address. As mentioned earlier, precondition 4 can be met trivially for systems with *sigreturn* gadgets at fixed locations. On other systems, the same restrictions as for precondition 3 apply.
The attack flow with this flavor of SROP looks as follows [BB14]:

1. The stack is overwritten by the attacker using a vulnerability.
2. When the program executes a *return* instruction, the program flow is redirected to the *sigreturn* gadget by an overwritten instruction pointer stored on the stack.
3. The *sigreturn* call loads all register values from the stack. This way, the registers can be prepared for using an *execve* system call with pointers to the attacker controlled data as arguments.
4. The application selected by the attacker can then be executed by using the gadget to place another system call. This way, the attacker can for example spawn a shell.

If the attacker wants to perform more complex operations than spawning a shell or starting some other program, step 3 can be changed to call *mprotect* to make the user controlled data executable. By setting the instruction pointer in the signal frame to the user controlled data, arbitrary code can then be executed.

2.2 SROP Chain

This attack is far more complicated, but has other preconditions which can make it more suitable for some situations. Because of differing system call numbers, it only works as depicted with x86_64 Linux versions. The attack should also work with small modifications on x86, OSX BSD and ARM in thumb mode but additionally needs a way to set the appropriate register (*EAX/RAX/R7*) to 3 for a read call or to the correct value for a *sigreturn* call, either with a system call or an additional gadget [W3]. The exact reason for this will be made clear following the presentation of the attack. The preconditions are as follows [BB14]:

1. The attacker needs a way to control the instruction pointer, e.g. through a *return* instruction on stack overwritten by the attacker.

2. The stack pointer has to be located on data controlled by the attacker where adding *NULL* bytes is possible.
3. The address of a *syscall* and *return* gadget has to be known.
4. The *RAX* register has to be set to 15 at the beginning.
5. The location of any writable page in memory has to be known.
6. The data sent to a pipe, file or socket with a known file descriptor number is controlled by the attacker.

For the preconditions 1 to 3, the same restrictions as in the previous section apply but the precondition 4 is far easier to match. Since the content and location of this page don't matter, any random page in the data section of the program can be picked. Even if the program changes and the data section moves a bit, the attack is not affected as long as the selected page is still a valid writable page. This way, the exact same attack can work for different versions of a program and can in some cases even work for other programs, as long as there is a writable page at the same place. Precondition 5 can be more complicated to meet depending on the situation. In cases where the attacker has access to the standard input stream of the program, for example when exploiting a binary with a *setuid* bit to escalate privileges, this is trivial. On the other hand, when exploiting a remote code execution vulnerability on a web server which may have hundreds of open connections, this can be far more difficult. The difficulty is mitigated a little by the fact that file descriptor numbers are always allocated in sequence and that the data can be sent over multiple connections. For example, if it is known that probably less then 50 users are connected to a web service, one could pick the file descriptor number 50 and send the needed data over 50 connections. If the initial assumption is right, one of the connections should get the file descriptor number 50 which would lead to success.
To perform this attack, a chain of *sigreturn* and other system calls is combined with user supplied data read from the file descriptor to execute arbitrary code. The concrete steps for this are as follows, a graphical overview can be seen in [BB14].

1. Using a stack overflow vulnerability, the return pointer of a function is replaced with a pointer to the *sigreturn* gadget. If only a *syscall* gadget is available, the RAX register has to be set to 15 so that the *sigreturn* system call is executed. A signal frame filled only with zero values except the following content has to be placed right beneath it.
    a) Stack pointer(*RSP*): Address of the writable page.
    b) Instruction pointer(*RIP*): Address of the *syscall* and *return* gadget.
    c) Akkumulator register(*RAX*): System call number for *read*.
    d) *RDI* register: File descriptor number, where to read from.
    e) *RSI* register: Address of the writable page, where to write to.
    f) *RDX* register: 306, number of bytes to read.

2. After the *sigreturn* gadget is executed, the *syscall* and *return* gadget is executed again because of the instruction pointer set in the frame. Because of the values in the registers, the system call *read* is executed which can write up to 306 bytes from the data received on the file descriptor to the writable page. The second part of the exploit is now sent over the file descriptor. This part contains the address of the *syscall* and *return* gadget three times, then a signal frame and following this, one can include arbitrary code or information. The amount of data received has to be 306 bytes for the next steps to work.

3. After having received the data, the *read* function returns with the number of bytes read in the *RAX* register. Our writable page is populated with the data that was sent and the stack pointer points to the beginning of our writable page. This leads to another call to the *syscall* function. Because of the value 306 returned by the *read* function, the system call *syncfs* is executed. This system call should always return with 0, the *syscall* number of *read* on x86_64, in the *RAX* register, as long as the file descriptor is still valid.

4. Another system call is executed due to the pointer on the stack. The value 0 in the *RAX* register leads to another *read* call. This time, the data sent is unimportant, the only thing that matters is that exactly 15 bytes are sent. This leads to *RAX* containing the value 15, the system call number for *sigreturn* .

5. Our last pointer initiates yet another system call. This time, the value 15 leads to another *sigreturn* call. The content of the following signal frame can be chosen according to ones need. One good option is executing an *execve* call with pointers to the data below the signal frame as arguments, another interesting option is calling *mprotect* to make the data below the frame executable.

6. The selected system call is executed. Either we are already done because we initiated *execve* directly, or we now have attacker controlled executable code at a known location which makes further exploitation trivial.

As could be seen, this technique uses *syncfs* to set the *RAX* register to 0 for a *read* call which finally enables the attacker to perform a *sigreturn* call. Therefor, this method is more difficult on other systems, since the *read* system call is executed with the value 3 on most. Of course if a *sigreturn* gadget is available, the *syncfs* and *read* call can be skipped altogether and the rest of this technique should work on all common systems. This is especially interesting on Android versions below 5.0 since they mostly use the ARM Linux kernel versions below 3.11 which maps a *sigreturn* call at a known address for all processes [BB14, An].

2.3 Conclusion

As could be seen in this section, using *sigreturn* during attacks is quite powerfull. With the use of a single gadget, arbitrary code can be executed with preconditions which can

be fulfilled easily in some cases. What makes it especially interesting is, that as long as the preconditions are met, the internal structure of an application almost doesn't matter for SROP based attacks.

3 Asterisk Exploit

The Asterisk Exploit is made possible by a stack consumption vulnerability with the CVE number *"2012-5976"* [Vua], for which a ROP remote execution attack was presented in [As]. The vulnerability being used lies in the HTTP Asterisk Management Interface and is due to an *alloca* call with a remotely supplied Content-Length value when receiving a HTTP POST header. The difficulty in exploiting it relies in the fact, that the allocated buffer is filled with the *fgets* method, which makes it impossible to wrap the stack pointer around to a location higher on the stack, since *fgets* doesn't allow values greater than *0x7FFFFFFF*. This means, that it isn't possible to alter the stack of the thread which created the buffer. The way around this is to create a second connection to the management interface to start an additional thread. As long as no other thread was created in between, the distance between the stack of the first connection to the stack of the second connection is defined in *AST_STACKSIZE* and is exactly *0x3C000*. By using a greater value, one can create a buffer on the first connection which contains the stack of the second connection. The steps to overwrite the return instruction pointer and other parts of the stack are therefor as follows [As]:

1. Creating two remote connections to the Asterisk Management Interface.
2. Sending a POST header with the Content-Length set to *0x3C900* to the first connection.
3. Sending approximately *0x700* bytes of data to the first connection.
4. Sending a POST header with the Content-Length set to *0x200* bytes to the second connection.

By sending the right value of bytes, the buffer pointer of the first thread can be placed directly above the return instruction pointer of the *_IO_getline_info* method of the second thread. Placing it there is essential, since the *RAX* register then contains the number of bytes read when our payload is executed, which is necessary to fulfill precondition 4 mentioned in section 2.2. Now it is time to send the first SROP payload:

3.1 First SROP Payload

To create the first SROP payload, we first need to know the address of a *syscall* and *return* gadget, the location of a writable page in memory and the number of a file descriptor on which data can be sent.
Since Asterisk was deployed on a server with Debian 7.9, getting the address of a *syscall*

and *return* gadget is trivial, as this distribution still uses a *vsyscall* page to provide some system functions [BB14]. If this wasn't the case, we would have to find a *syscall* and *return* gadget in the binary. This can be done with tools such as ROPgadget [Gi].
To find a writable page, we just have to use the *readelf* utility on the Asterisk binary [re]. This way, the different sections of the binary are displayed. Any page in the *.data* and *.bss* section can be used.
The final requirement is a valid file descriptor number to which we can write. While no usable file descriptor number is known, we can work around this as was described in section 2.2 by using many connections to the server. These connections should be opened before executing the first payload as the *read* call will return with an error if it is applied to a file descriptor number which isn't opened yet.
The first payload can now be created as was explained in section 2.2 with the values we selected before. To create the concrete signal frame, one can either use the SROP library referred to in [MP15] or, since it is the same as a context frame, one can use the *ucontext* struct which is defined in *ucontext.h*.
After having sent the first payload to the first connection, the first payload can be executed by sending 15 bytes to the second connection and closing it.

3.2 Second Payload

The second thread should now be waiting on 306 bytes of data on the connection to which the selected file descriptor number belongs. To continue the exploit, we have to create the second payload as described in section 2.2. The values for the signal frame and the data below it can be chosen freely to execute any system call we like. For further exploitation, using *mprotect* to make the writable page executable was chosen.
One problem building the second payload is, that only 306 bytes are available. This is critical, since the *ucontext* struct alone has a size of 944 bytes. This problem can be solved by building a struct which only includes *uc_flags*, **uc_link*, *uc_stack*, *gregs* and *fpregs* from the *ucontext* struct since the rest of the struct is not needed for this application. This gives us a size of 232 bytes for the signal frame. Together with the 3 pointers to the *syscall* and *return* gadget and a pointer to our own data to execute it, this leaves us with 42 bytes for our own code. Since this isn't enough for most applications, a shellcode was added which simply reads from the file descriptor number again and appends the result to the already existing code. This way, an arbitrary amount of code can be executed as long as it fits in the *.data* or *.bss* section of the binary.
After the second payload was constructed, it still has to be sent to the right connection and be read by the command issued by the first payload. One thing which makes this more difficult is, that each connection we created spawned an own thread which tries to read from it already and which closes the connection as soon as it received a request and responded to it. One option to get a window of opportunity in which our payload can read from the connection without being disturbed is to send a login request to the owner of the connection. While the original owner checks if the account information is valid, one has enough time to communicate with our payload.
To finish our exploit, we have to go through all connections we opened and send every

one a login request followed by our second payload, fifteen random bytes and finally the shellcode we want to execute - in this case a reverse shell connection to another server taken from [Li].

4 Glibc Exploit

In early 2016, a stack based overflow vulnerability with the CVE number *"2015-7547"* concerning DNS address resolution in the widely used glibc was disclosed [Vub]. The exploitation path for this vulnerability was outlined in the patch notes on a glibc mailing list and looks as follows [PA]:

- The attacker needs to be able to control the DNS answers received by the program.
- The program to be exploited must send out two DNS queries at once, e.g. for the A and AAAA records.
- The attacker needs to respond to both DNS queries. The first response needs to be a valid DNS response with a length of 2048 bytes to completely fill the allocated stack buffer. The second response now triggers a malloc call to create a new buffer with 65535 bytes since the response buffer is allready full. To trigger the bug, the second response has to be flawed in a way that gets glibc to retry the query.
- Glibc then sends two more requests and reuses the 2048 bytes big buffer placed on the stack to save the corresponding responses. Due to a bug, the size of the buffer is now wrongly specified as the size of the heap buffer, 65535 bytes.
- The attacker can now send an overlong response to the program which leads to an overflow of the stack buffer.

This way, a Denial of Service attack against any target still using a vulnerable version of glibc is easily created.
Remote code execution, as tested on Debian 7.9 using the newest glibc package before the bug fix, isn't quite as simple. This is due to the fact that the address of the stack buffer used to save the result is saved on a part of the stack which is overwritten. Setting this address to *NULL* leads to an assert failure and writing any value differing from the initial one to it leads to a free call on this address, which causes an exception if the address doesn't point to a valid freeable memory block. Therefore, a stack memory disclosure vulnerability is also needed to be able to enable code execution. This means that the only additional preconditions needed for an address based SROP attack are a method to set *RAX* to 15 and a *syscall* gadget. These can be fulfilled as described in section 2.1.
To create a remote code execution exploit, a simple exploitable program was implemented. This program leaks a stack address to the attacker and then tries to resolve an address using the getaddrinfo method of glibc as shown in [fj]. To make the exploitation easier, an attacker controlled server is set as the primary DNS server of the server which is to be attacked although this could be bypassed if the attacker is able to intercept the messages sent by the vulnerable program through some other way. The attacker can now trigger the

vulnerability as previously described. The only restriction is, that the memory location containing the address of the initial response buffer has to be filled with the same value as before. This can be done by calculating the address using the leaked stacked address and some fixed offset. Finally, arbitrary code can be executed using SROP as shown in section 2.1.

5 Reusability of SROP Exploits

To demonstrate the reusability of SROP exploits, two simple exploitable programs were created, one for address based exploits and one for SROP chain exploits. These programs were built in a way to fulfill all conditions required for the specific exploit.

5.1 SROP Chain based Exploit

This program contains a large global char array, so that enough writable space is available. After starting, it simply executes a function with a return value of type long. This function first allocates a variable from type long and a char array of size 10 on the stack, then asks how many bytes it should receive and uses *read* to write the number of bytes specified to the array. Since the number of bytes that should be written is provided by the user and isn't checked by the program, a stack overflow can be caused trivially by sending it a number greater than the size of the array. Finally, the function returns the value of the previously allocated long variable.
After taking a look at the previous Asterisk exploit, it was determined that the two SROP Chain payloads only depend on three parameters:

- The address of the *syscall* and *return* gadget.
- The address of the writable page.
- The file descriptor number used to communicate with the exploit.

The SROP payload creation instructions where therefor extracted from the Asterisk exploit to a library which then contains two simple to use functions to create the first two payloads with the aforementioned parameters. Since the final shellcode also doesn't depend on the application, this part could also be copied without altering it.
The address of the *syscall* and *return* gadget and of the writable page where acquired in the same way as for the Asterisk exploit. Since the exploit communicates with the exploitable program over the standard input stream, the file descriptor number was simply set to *STDIN_FILENO*. This information was used to create the two SROP payloads.
The only thing left was adding some bytes before the first SROP payload, so that the payload starts at the return address of the function, while taking care that the long variable on the stack is overwritten by the value 15 to initiate a sig*return* call.
The exploit therefor only has to do the following steps:

1. Generate the two SROP payloads using the SROP library and the supplied addresses and file descriptor number.
2. Start the exploitable program.
3. Send the first payload together with the additional bytes.
4. Send the second payload.
5. Send 15 random bytes.
6. Send the final shellcode.

This shows that quite a lot of the code of a SROP chain exploit can be reused from one application to another. While, depending on the vulnerability, overwriting the return instruction pointer and the part below it on the stack may still be quite complicated, the rest of the exploit is easy, provided we have the following conditions:

- One can communicate freely with a known file descriptor number
- Finding the two required addresses isn't complicated by some defence mechanism.
- There is a way to set the *RAX* value to 15, either with a gadget or by exploiting a function which returns a value from the stack using this register.

5.2 SROP Exploit with known address

The same program was reused to demonstrate an SROP exploit with a known address with the only difference being, that the global buffer was removed and a stack memory address disclosure was added. As with the Chain based attack, all the SROP specific code from the glibc attack was exported to the library, since it only depends on the location of a *syscall* gadget and the address of the stack at which the exploit is located. Reusing the known address based SROP code from the glibc exploit worked without any modifications and due to the few restrictions, this should be the case for all vulnerabilities which give the same preconditions.

5.3 Conclusion

Compared to ROP based exploits, where, after overwriting the stack, one still has to create a ROP chain, which varies strongly from one program to another and which can be quite complicated and long depending on the available gadgets, using SROP is far simpler. When the preconditions of one of the two SROP flavors are met, the entire ROP Chain can be replaced by some reusable SROP based code which can be automatically generated by a library.

6 Impact of Defence Mechanisms

This section will examine how critical some defence mechanisms are for the usage of SROP exploits. The examined mechanisms are the default implementation of address space layout randomization (ASLR) on Linux, position independent executables (PIE) and Oxymoron.

6.1 Address Space Layout Randomization

ASLR is a technique to map each part of a program to a random address on start. It is activated by default on Linux since the kernel version 2.6.12 released in 2005. The problem with the Linux implementation is, that for normal executables, while some parts of the program, like external libraries, are added at random addresses, the program code and the data sections still always are at the same location [Ho]. Since the gadget requirements of SROP are quite low and the data sections can be used as writable pages, ASLR shouldn't affect SROP Chain exploits in most cases.

6.2 Position Independent Executables

By compiling an application as a PIE, one can get Linux to put every section of the program at a random address. This makes SROP attacks far more difficult, since the addresses of writable pages and gadgets are not fixed. But SROP attacks, like ROP attacks, might still be possible if the program also has a memory disclosure vulnerability [As].

6.3 Oxymoron

Oxymoron improves on ASLR by splitting every part of the program at the page level and placing every page at a random location. With Oxymoron, finding out an address through a memory disclosure only helps locating one single page of a section, which makes ROP attacks very difficult, since in most cases, a single page won't contain all the gadgets needed to mount an attack. Because SROP attacks require only one or two gadgets and the only other address information needed is the location of some writable page, an SROP attack on Oxymoron protected executables should still be possible in some cases [MP15].

7 Conclusion

This paper demonstrates how SROP exploits can be created and that SROP makes stack overflow based attacks easier compared to ROP attacks when some conditions are met. This is especially true since large parts of the exploit may be reused between different applications which is shown by creating an application independent SROP library for the

x86_64 architecture.
While some existing protection mechanisms can make SROP exploitation far more difficult to perform, they still impede SROP attacks less than ROP attacks because of the lower gadget requirement. This makes SROP a real threat and highlights the need for SROP mitigation techniques for Unix systems like the ones proposed in [BB14].

References

[An] Android Versions - eLinux.org. http://elinux.org/Android_Versions. Accessed: 2016-02-16.

[As] Asterisk Exploit — Exodus Intelligence. http://blog.exodusintel.com/tag/asterisk-exploit/. Accessed: 2016-02-15.

[BB14] Bosman, Erik; Bos, Herbert: Framing Signals â A Return to Portable Shellcode. IEEE Symposium on Security and Privacy, pp. 243–258, 2014.

[fj] fjserna/CVE-2015-7547. https://github.com/fjserna/CVE-2015-7547/blob/master/CVE-2015-7547-client.c. Accessed: 2016-04-25.

[Gi] Github JonathanSalwan/ROPgadget. https://github.com/JonathanSalwan/ROPgadget/. Accessed: 2016-02-15.

[Ho] How Effective is ASLR on Linux Systems? http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/. Accessed: 2016-02-15.

[Li] Linux/x86-64 - Reverse TCP shell - 118 bytes. http://shell-storm.org/shellcode/files/shellcode-857.php. Accessed: 2016-02-15.

[MP15] Mithra, Zubin; P., Vipin: Evaluating the theoretical feasibility of an SROP attack against Oxymoron. International Conference on Advances in Computing, Communications and Informatics, pp. 1872–1876, 2015.

[PA] PATCH CVE-2015-7547 — glibc getaddrinfo() stack-based buffer overflow. https://sourceware.org/ml/libc-alpha/2016-02/msg00416.html. Accessed: 2016-04-25.

[re] readelf - GNU Binary Utilities. https://sourceware.org/binutils/docs/binutils/readelf.html. Accessed: 2016-02-15.

[Vua] Vulnerability Summary for CVE-2012-5976. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5976. Accessed: 2016-02-15.

[Vub] Vulnerability Summary for CVE-2015-7547. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-7547. Accessed: 2016-04-25.

[W3] W3Challs Syscall tables. https://w3challs.com/syscalls/. Accessed: 2016-02-16.