Packed, Printable, and Polymorphic Return-Oriented Programming

Kangjie Lu[1,2], Dabi Zou[1], Weiping Wen[2], Debin Gao[1]

[1] School of Information Systems, Singapore Management University, Singapore
`{kjlu, zoudabi, dbgao}@smu.edu.sg`
[2] School of Software and Microelectronics, Peking University, China
`weipingwen@ss.pku.edu.cn`

Abstract. Return-oriented programming (ROP) is an attack that has been shown to be able to circumvent $W \oplus X$ protection. However, it was not clear if ROP can be made as powerful as non-ROP malicious code in other aspects, e.g., be packed to make static analysis difficult, be printable to evade non-ASCII filtering, be polymorphic to evade signature-based detection, etc. Research in these potential advances in ROP is important in designing counter-measures. In this paper, we show that ROP code could be packed, printable, and polymorphic. We demonstrate this by proposing a packer that produces printable and polymorphic ROP code. It works on virtually any unpacked ROP code and produces packed code that is self-contained. We implement our packer and demonstrate that it works on both Windows XP and Windows 7 platforms.
keywords: Return-oriented programming, packer, printable shellcode, polymorphic malware

1 Introduction

Return-oriented programming (ROP) [23] and its variations [6–8,12,15,16] have been shown to be able to perform arbitrary computation without executing injected code. It executes machine instructions immediately prior to return (or return-like [7]) instructions within the existing program or library code. Both the address words pointing to these instructions and the corresponding data words are usually called gadgets. Since ROP does not execute any injected code, it circumvents most measures that try to prevent the execution of instructions from user-controlled memory, e.g., the $W \oplus X$ [1] protection mechanism.

Although ROP has been shown to be powerful in circumventing the $W \oplus X$ protection, it was unclear whether it can be as powerful as non-ROP malicious code in many aspects, e.g., be packed to make static analysis difficult, be printable to evade non-ASCII filtering, be polymorphic [2,11] to evade signature-based detection, etc. Investigation into these topics is important as the advances could make ROP shellcode much harder to detect.

In order to find useful machine instructions, ROP usually expands the search space from the executable binary to shared libraries. Although ROP has been shown to be able to perform arbitrary computation on many platforms, intuitively there is little flexibility in constructing an ROP shellcode since there are limited candidates of such machine instructions. Therefore, it is unclear the extent to which ROP shellcode can be made polymorphic, i.e., ROP shellcode that looks different but perform similar functionality.

Making printable ROP shellcode is even more challenging. ROP shellcode is mainly composed of addresses[3], e.g., `0x0303783e`, while the range of ASCII printable characters is between `0x21` and `0x7e`. Since printable characters only account for roughly 36.7% of all characters, if useful gadgets are uniformly distributed across the entire address space, then roughly $(36.7\%)^4 \approx 1.8\%$ of these gadgets (and their corresponding machine instructions) can be used.

We propose using a packer to make ROP shellcode printable and polymorphic. Our proposed packer is inspired by techniques that make traditional shellcode printable (e.g., [21]) where alphanumeric opcodes (e.g., `pop ecx` has an opcode `0x59` which is the ASCII code of the character `Y`) are used to transform non-printable shellcode into alphanumeric shellcode. Each non-printable 4-byte address is represented by two 4-byte printable addresses by our packer. The packed ROP shellcode takes the two printable addresses and performs arithmetic operations on them to restore the original non-printable address. Since there are many options in choosing the two 4-byte printable addresses for any given non-printable address, we are able to construct polymorphic printable shellcode. We also propose a two-layer packer to reduce the size of the packed code by reusing (looping) gadgets that perform arithmetic operations[4]. The packed code constructed is self-contained, i.e., it does not require an external loader to execute.

We implement our two-layer packer and use it to pack two real-world ROP shellcode on both Windows XP and Windows 7 platforms. All the machine instructions used are from common libraries. We demonstrate that the packed printable shellcode works well on both Windows XP and Windows 7 platforms. As an extension, we demonstrate another use of our ROP packer as a polymorphic converter to make the resulting packed code immune to signature-based detection. We also show that our packer works not only on ROP using return gadgets, but also ROP using non-return gadgets [5, 7].

2 Related work

Shacham et al. proposed Return-Oriented Programming (ROP) [23]. ROP uses a large number of instruction sequences ending with `ret` from either the original program or libc, chains them together to perform arbitrary computation.

[3] Besides addresses, there are also constants and junk data in ROP shellcode.

[4] It might not be appropriate to call it a packer as most packed code produced by existing packers performs decompression or decryption. Our packed code, instead, decodes printable addresses into the original non-printable ones. Therefore, it might be more appropriate to describe our packed code as containing a decoder. We use the term packer mainly because of our second-layer decoding which dramatically reduces the size of the packed code. It makes our solution similar to multi-layer packers.

ROP is also extended to many platforms such as SPARC [6], ARM [16], Harvard [12], and voting machines [8]. On the other hand, some researches are seeking to detect and prevent ROP attacks. Davi et al. [10] and Chen et al. [9] detect the ROP when the number of consecutive sequences of five or fewer instructions ending with a `ret` reaches a certain threshold. Buchanan et al. [6] and Francillon et al. [13] use the shadow return-address stack to defeat against ROP. Most recently, Onarlioglu et al. [19] propose G-Free, which is a compiler-based approach to eliminate all unaligned free-branch instructions inside a binary executable and prevent aligned free-branch instructions from being misused.

It is generally believed that ROP code needs to be carefully prepared and it is not clear to what extent variations can be made to it without changing the semantics. This raises a question as whether various attacking techniques proposed for malware can be used on ROP as well, e.g., polymorphic malware [26], packed malware [25], printable shellcode [21], etc.

Rix proposed a way to write IA32 alphanumeric shellcode [21] which uses some basic instruction sequences whose opcode is alphanumeric to transform two alphanumeric operands into one non-alphanumeric code. Others proposed different shellcode encoding approaches, e.g., UTF-8 compatible shellcode [14], Unicode-proof shellcode [18], etc. Most recently, Mason et al. proposed to automatically produce English shellcode [17], transforming arbitrary shellcode into a representation that is superficially similar to English.

Unfortunately, none of these approaches is based on ROP, in which the register `esp` has a special usage as a global state pointer (just like `eip`) to get the address of the next group of machine instructions. Existing approaches of making shellcode printable changes the value of `esp` only with side effect, and therefore are not suitable for making ROP printable.

3 Overview

We first present a one-layer packer (resulting in long shellcode) and an overview of our two-layer packer (with an additional decoder to make shellcode shorter).

3.1 One-layer Printable Packer for ROP

Many useful instructions in ROP have non-printable addresses. Since they are hard to find in general, simply not using them has a large negative impact on what ROP could perform. As shown in Section 1, only 1.8% of the addresses are printable assuming that useful instructions are uniformly distributed across the entire address space. Our solution is to transform these non-printable addresses into printable bytes and then use a decoder to get back the original addresses.

However, the decoder in the packed code has to be implemented by printable gadgets, which dramatically limits the instructions we can use. We need to find those with printable addresses that are able to decode *any* addresses, since we want to design a packer that works on *any* unpacked ROP shellcode.

To handle this difficulty, we use multiple (two to three in our experiments) printable 4-byte codes to represent a 4-byte non-printable address. For example, a gadget with a non-printable address `0x7d59869c` can be represented by two printable codes `0x2d30466c` and `0x50294030` with an operation of addition. Fig 1 shows the idea (the actual shellcode is slightly more complicated).

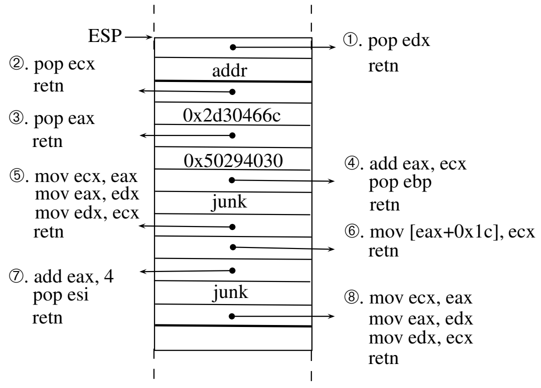


Fig. 1. One-layer packer

Fig 1 shows two parts of the packed shellcode. The first part consists of one gadget ① and an address `addr`. `addr` points to the location (in data segment) where the decoded (non-printable) addresses will be written, and gadget ① loads it into `edx`. The second part consists of gadgets and corresponding data for decoding the first non-printable address. Subsequent parts look similar to the second part, which are for decoding other non-printable addresses. Next, we look into the details of decoding one non-printable address (middle portion of Fig 1).

Gadget ② and ③ first load the two printable 4-byte operands `0x2d30466c` and `0x50294030` into `ecx` and `eax`, respectively. Decoding is performed by gadget ④ to add the two printable operands and store the result `0x7d59869c` to `eax`. Note that here we use gadget ④ with side-effects [28] of popping one 4-byte code, and that is why we have to add a 4-byte junk between gadget ④ and ⑤.

Next, we move the result `0x7d59869c` to `ecx` with gadget ⑤ (again, with side effects), and subsequently to the location pointed to by `addr` using gadget ⑥. We then add 4 to `eax` (gadget ⑦) so that it points to the next writable address beginning at `addr`, and load it to `edx` (gadget ⑧). With this, we can move on to decode the next non-printable addresses, after which we use stack pivot [30] to make `esp` point to the decoded original ROP shellcode and execute it.

This one-layer packer is simple, but the packed code is long. As shown in Fig 1, we need 11 4-byte codes to decode one 4-byte non-printable address. Some non-printable addresses, e.g., `0x0303783e`, might need a longer decoder as it is impossible to find two printable 4-byte addresses whose sum equals to it. Therefore, we need a better decoder to shorten the packed code.

3.2 Two-layer Printable Packer for ROP

Analyzing the packed code shown in Fig 1, we realize that only the two 4-byte operands to be added are unique in each round of the decoding process. The other nine 4-byte codes are either addresses of instructions which are the same in each round of decoding, or junk. Therefore, a key idea of reducing the size of the packed code is to separate data (the two printable 4-byte operands to be added) from the decoding routine and to put the decoding routine into a loop. If this can be done, the size of the packed code will be two times the original shellcode (each non-printable address is represented by two printable 4-byte operands) plus the size of the decoder (hopefully fixed-size).

Unfortunately, we cannot find all the required printable gadgets to implement the loop.[5] Our solution is to have two layers of decoders where the second layer, denoted dec^2, decodes the original ROP shellcode (possibly using non-printable gadgets), and the first layer, denoted dec^1, decodes dec^2 (see Fig 2).

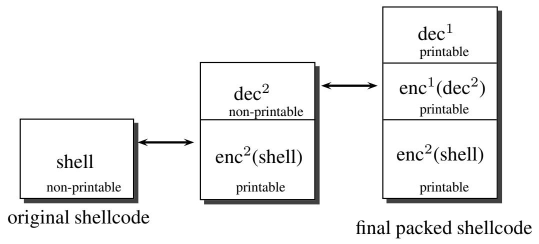


Fig. 2. Two-layer packer

A by-product with the two-layer design is the flexibility in choosing the 4-byte operands for decoding, and therefore polymorphism of the resulting packed code. In the one-layer design discussed in Section 3.1, we only have flexibility in choosing two 4-byte operands (which adds to the original non-printable address) and the junk. There is little flexibility to some gadgets shown in Fig 1 and therefore one could easily find reliable signatures to the packed code. Our two-layer design introduces a new layer and more opportunities of polymorphism. Section 6.2 further discusses this and limitations of our approach.

4 Two-Layer Encoding and Degree of Polymorphism

Our two-layer packer enables the conversion from left to right as shown in Fig 2. enc^2 takes as input the original shellcode shell and produces two outputs, dec^2 and $\mathsf{enc}^2(\mathsf{shell})$. dec^2 goes through another encoding process which outputs dec^1

[5] This confirms our earlier conjecture that useful instructions with printable addresses are difficult to find in ROP.

and $\mathsf{enc}^1(\mathsf{dec}^2)$. In this section, we focus on the encoding processes. Section 5 shows how the decoders work to enable the conversion from right to left in Fig 2.

The encoding processes of these two layers are similar in that both use 4-byte printable operands to represent non-printable addresses in shell and dec^2. A difference that dec^1 (output of the second encoding) has to use printable gadgets while dec^2 (output of the first encoding) does not have to. Therefore it is more difficult to find gadgets to implement dec^1, while gadgets for dec^2 are easier to find.

Due to this additional restriction in implementing dec^1, we decide to use *three* 4-byte printable operands in $\mathsf{enc}^1(\mathsf{dec}^2)$ to represent a non-printable 4-byte in dec^2, while use only *two* 4-byte operands in $\mathsf{enc}^2(\mathsf{shell})$. Reason is simple — we do not manage to find printable gadgets whose arithmetic operation can represent any non-printable 4-byte address with two printable 4-byte operands, while we do manage to find printable gadgets performing `(op1 - op2) xor op3` which fulfills our requirements. Finding gadgets for dec^2 is easier, and in our experiment we use on that performs `((op1 << 1) + 1) xor op2 = i`.

Now, given a 4-byte input code `i` (most likely a non-printable address in a gadget), we need to automatically find the values of

– `op1`, `op2`, and `op3` such that `(op1 - op2) xor op3 = i` in $\mathsf{enc}^1(\mathsf{dec}^2)$; and
– `op1` and `op2` such that `((op1 << 1) + 1) xor op2 = i` in $\mathsf{enc}^2(\mathsf{shell})$.

To simplify our discussion, we assume that `i`, `op1`, `op2`, and `op3` are of one byte long. A small modification is needed when dealing with 4-byte codes to take care of the subtraction and shifting operations.

Our algorithm of finding the operands is simple. We first randomly assign a value from the range of printable bytes `[0x21, 0x7e]` to one of the operands. After one of the operands is chosen, we check if the chosen value makes it impossible for other operands to be printable. If yes, go back and choose a different value; otherwise, proceed to determine the next operand in the same way.

We have implemented the two encoders for both Windows XP and Windows 7. We assume that the address of the data segment of the vulnerable application is known. On Windows 7, we additionally assume that the base addresses of `ntdll.dll`, `kernel32.dll`, and `shell32.dll` are known, an assumption previous work on ROP also makes [6, 22, 23].

Finding operands in $\mathsf{enc}^1(\mathsf{dec}^2)$ We first randomly assign `op3` a value from the range `[0x21, 0x7e]` and calculate `op1 - op2 = i xor op3`. Note that since both `op1` and `op2` have to be printable, `op1 - op2` must fall into the range of `[0x00, 0x5d]` or `[0xa3, 0xff]`. If `i xor op3` falls outside of this range, we have to go back and choose a different `op3`. After `op3` has been chosen, we run the same algorithm to determine `op1` and subsequently `op2`[6].

[6] Some optimizations are possible in this process. For example, if $|(op1 - op2) < 0x7e$, we can randomly select $op1$ from $[0x21 + ixorop3, 0x7e]$; otherwise, we select $op1$ from $[0x21, (0x7e + ixorop3)\ AND\ 0xff]$. $op2$ can then be obtained by adding $ixorop3$ to $op1$. Similar optimizations can be used to calculate $op3$ and to find operands in enc^2. We do not discuss these optimizations further.

Finding operands in `enc`^{2}*(*`shell`*)* For `enc`^{2}, the first operand to be determined is `op2`. We randomly assign `op2` a value from the range `[0x21, 0x7e]` and calculate `(op1 << 1) + 1 = i xor op2`. Since `op1` has to be printable, `(op1 << 1) + 1` must fall into the range of `[0x43, 0xfd]` and the last bit must be 1. If `i xor op2` does not satisfy this condition, we have to go back and choose a different `op2`. After `op2` has been chosen, `op1` can be determined easily.

One may ask whether it is possible that no printable operands can be found satisfying the conditions. The answer is no, and that is because we specifically pick the arithmetic operations to avoid it. Table 1 and Table 2 show the number of operands that satisfy the conditions when `i` has a value from `0x00` to `0xff` for `enc`^{1} and `enc`^{2}, respectively. We see that no matter what `i` is, there are always a number of possible operands that satisfy the conditions.

Original byte `i`	Number of possible `op3`	Average number of possible `op1`
0, 1, 2, · · · , 33	61	1964
34, 35, · · · , 63	60	2884
64, 65, · · · , 95	63	3954
96, 97, · · · , 126	92	4372
127	93	4371
128, 129, 130	92	4279
131, 132, · · · , 159	91	4280
160, 161, · · · , 191	63	3891
192, 192, · · · , 220	59	2824
221, 222, 223	60	2823
224, 225, · · · , 255	60	1846
Weighted average	69	3244

Table 1. Number of possible operands for `enc`^{1}

Original byte `i`	Number of possible `op1`	Examples of `op1`
0, 2, · · · , 58, 60	30	{0}: 69, 71, · · · , 127;
1, 3, · · · , 59, 61, 62, 63	31	{1}: 67, 69, · · · , 127;
64, 66, · · · , 92, 94	16	{64}: 97, 99, · · · , 127;
65, 67, · · · , 93, 95, 96, 97, 98, 100, · · · , 124, 126	15	{65}: 97, 101, · · · , 127; {98}: 67, 69, · · · , 95;
99, 101, · · · , 125, 127	14	{99}: 69, 71, · · · , 95;
129, 130, · · · , 221, 222	46	{129}: 163, 165, · · · , 253;
128, 223, 224, · · · , 254, 255	47	{128}: 161, 163, · · · , 253;
Weighted average	35	N/A

Table 2. Number of possible operands for `enc`^{2}

Table 1 and Table 2 not only show that our two-layer packer is applicable of packing any unpacked ROP shellcode, but the degree to which polymorphism can be applied during the packing. For example, if `i = 127` in `enc`^{1}, there are 93 possible `op3` to choose from. Once `op3` has been chosen, there are (on average) 47 possible `op1` to choose from. That is, for this single byte in the original unpacked shellcode, we have about $93 \times 47 = 4371$ different ways of representing it. This shows the large degree to which polymorphism can be applied when running our encoders. Note that the above analysis applies to the last two portions in the final packed shellcode shown in Fig 2. We discuss this further in Section 6.

5 Decoders in Packed Shellcode

Having explained the encoding process, here we present the detailed implementation of the decoders. The two decoders dec^1 and dec^2 are similar in that they both have an initialization step to set up the environment and the actual decoding step. A difference is that dec^2 uses a loop while dec^1 does not.

5.1 Implementation of dec^1

The initialization of dec^1 first arranges some writable memory for temporary storage, and then initializes some registers. The decoding step loads the encoded dec^2 (i.e., $\mathsf{enc}^1(\mathsf{dec}^2)$) into registers and performs arithmetic operations to decode dec^2. Finally, control is transferred to the beginning of the decoded dec^2.

Initializing The purpose of initialization is to find the starting address of $\mathsf{enc}^1(\mathsf{dec}^2)$ and save it at a temporary storage. We do this to make it easy to load data of $\mathsf{enc}^1(\mathsf{dec}^2)$ into registers for decoding. As shown in Fig 3, there are four steps in the initialization in dec^1, which are clearly explained in the figure.

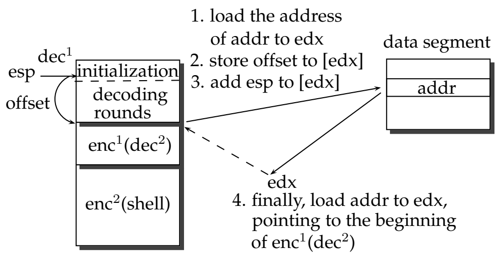
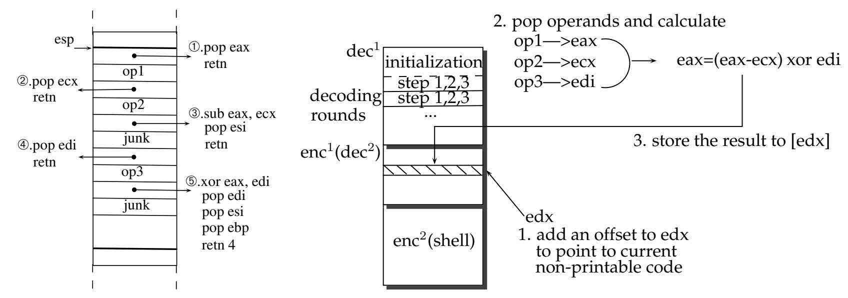


Fig. 3. Initialization of dec^1

Decoding $\mathsf{enc}^1(\mathsf{dec}^2)$ As shown in Fig 4a, the decoding is done by first loading the three 4-byte operands and then calculating `(op1 - op2) xor op3`. There are two types of data in $\mathsf{enc}^1(\mathsf{dec}^2)$, un-encoded data which corresponds to printable addresses in dec^2 and place-holders that are printable and random for non-printable addresses in dec^2 (enc^1 is discussed further in Section 4). The former can be left untouched, while the latter needs to be overwritten by the decoding routine (including data) in dec^1 (see Fig 4b).



(a) Arithmetic operations (b) Decoding

Fig. 4. Decoding in dec^1

There are three steps in the decoding. First, we locate the next non-printable address in enc^1(dec^2), and add its offset to `edx`. Second, arithmetic decoding (see Fig 4a) is performed, the result of which is stored in `[edx]` in the last step.

5.2 Implementation of dec^2

Similar to dec^1, dec^2 also has an initializing step and a decoding step. However, dec^2 is slightly more complicated due to the use of a loop.

Initializing The purpose of the initialization is similar to that in dec^1. However, we need some more temporary storage in dec^2, since gadgets in the loop are separated from data (enc^2(shell)). We need pointers to point to the data (`addr1` and `addr2` in Fig 5 for reading and writing, respectively) and pointer to point to the starting of the loop (`addr3` in Fig 5).

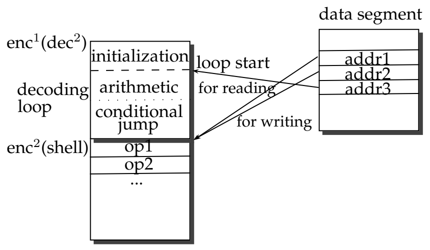
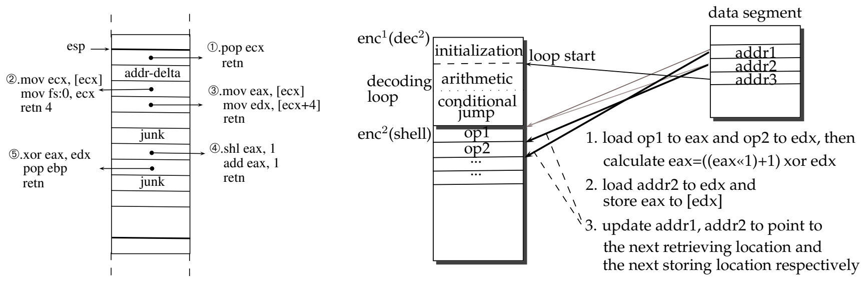


Fig. 5. Initialization of dec^2

By adding the offset of enc^2(shell) to the current value of `esp`, we obtain the starting address of enc^2(shell) and store it in the temporary storage `addr1` and `addr2`. `addr1` is used to hold the address from which operands are read for decoding, while `addr2` is used to hold the address to which the decoded addresses are stored. We also calculate the starting address of the decoding loop and store it in `addr3` to which execution jumps at the end of every loop.

Decoding enc^2(shell) Recall that dec^2 might use non-printable gadgets. We are more flexible in choosing the arithmetic operations in this decoding, and do not have to go for three operands as in enc^1. Again, we aim for arithmetic operations that can represent any non-printable 4-byte address with two printable 4-byte operands, and choose to use `((op1 << 1) + 1) xor op2` as shown in Fig 6a. See Section 4 for discussions on the choice of this arithmetic operation, the applicability of it, and the polymorphism of the resulting data.



(a) Arithmetic operations (b) Decoding

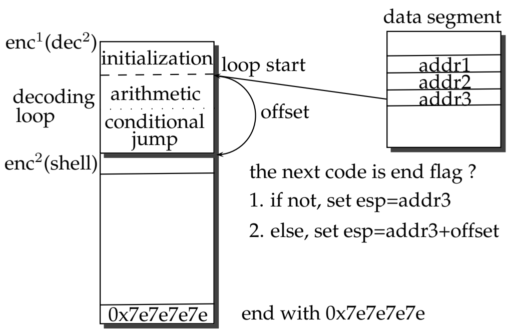
Fig. 6. Decoding in dec^2

In order to decode enc^2(shell), we first load the two 4-byte operands pointed to by `addr1` to `eax` and `edx` (indirectly, as shown in Fig 6b, due to unavailability of gadgets that can do this more directly), and then perform the arithmetic operations to calculate the non-printable address in shell. Second, we load the value of `addr2` to `edx` and save the decoded address to `[edx]`. After that, `addr1` is updated with an offset of 8 (two operands) while `addr2` is updated with an offset of 4, and control is transferred back to the beginning of dec^2 (pointed to by `addr3`) to decode the next address. Fig 6b shows this process.

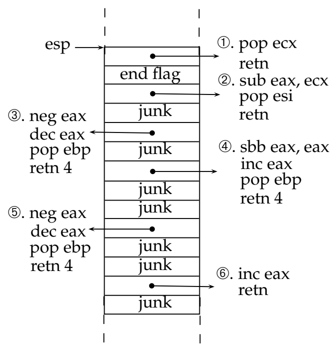
Note the addition step in dec^2 to perform a conditional jump. To signal the end of the decoding, we append a special word `0x7e7e7e7e` to the end of enc^2(shell) as a stop indicator. Fig 7a illustrates the idea.

5.3 Gadgets used in our implementation

In this subsection, we describe the instruction sequences and the corresponding gadgets we use to construct our two-layer packer. Automatically searching for printable gadgets is relatively simple. We just modified the Galileo Algorithm [23] to add an additional condition on the address. We search for gadgets



(a) Conditional jump in dec^2



esp
end flag
junk
junk
junk
junk
junk
junk
junk
①. pop ecx retn
②. sub eax, ecx pop esi retn
③. neg eax dec eax pop ebp retn 4
④. sbb eax, eax inc eax pop ebp retn 4
⑤. neg eax dec eax pop ebp retn 4
⑥. inc eax retn

(b) Gadgets to implement conditional jump

Fig. 7. Conditional jump

on both Windows XP SP3 (x86) and Windows 7 Ultimate (x86), and found all the printable and non-printable gadgets needed for constructing `dec`^{1} and `dec`^{2}.

Gadgets we use are from common shared libraries. For Windows XP, all the gadgets we use are from `shell32.dll` and `msctf.dll` with base addresses `0x7d590000` and `0x74680000`, respectively. Windows 7, on the other hand, uses ASLR [3, 4, 24, 27, 29] where the base addresses of libraries are randomized after every restarting. We assume that the base addresses of `ntdll.dll`, `kernel32.dll` and `shell32.dll` are known (of values `0x77530000`, `0x76710000` and `0x768e0000`, respectively in our experiment), an assumption previous work on ROP also makes [6, 22, 23]. Note that `ntdll.dll` and `kernel32.dll` have printable addresses which are used in `dec`^{1}, while `shell32.dll` has a non-printable address and therefore is used in `dec`^{2} only.

To describe the instruction sequences we found and how to build the gadgets, we take Windows 7 as an example and discuss the gadgets we use in our two-layer packer, with a focus on printable gadgets.

Basic gadgets Gadgets to load and store data are relatively easy to find even when we limit ourselves to printable gadgets. We use `pop` to load constants from the stack, and use `mov` to load data from other memory locations as well as to store data at memory locations. Gadgets that perform arithmetic operations are also easy to find as discussed in Section 5.1 and Section 5.2

To get the address of the stack, we need some `esp` related instruction sequences to store the value of `esp` to a register or a memory location. In the conditional jump in `dec`^{2}, we also need the “stack pivot” instruction sequences. Table 3 shows some examples.

Gadgets in dec^1 Gadget in `dec`^{1} need to have printable addresses. Fortunately, we manage to implement it with the basic gadgets described in Section 3.

Purpose	Instruction	Relative address	Printable	Library
Loading/storing data	`pop eax`	0x000a6656	Y	kernel32.dll
	`mov edx, [ecx+4]`	0x00057a4f	Y	ntdll.dll
	`mov [edx], eax`	0x0004662a	Y	ntdll.dll
Arithmetic operations	`shl eax, 1`	0x00034986	N	ntdll.dll
	`sub eax, ecx`	0x000c632b	Y	ntdll.dll
	`xor eax, edi`	0x000b3f46	Y	kernel32.dll
	`xor eax, edx`	0x0005ac24	N	ntdll.dll
esp related	`add [ecx+0x7760cc7c], esp`	0x00072b4d	Y	ntdll.dll
	`adc [ecx+0x4fc0007e], esp`	0x00055c5b	Y	kernel32.dll
	`mov esp, [ecx+0xd8]`	0x00004eef	N	ntdll.dll
	`xchg esp, eax`	0x0009f9d2	N	ntdll.dll

Table 3. Basic gadgets used in dec^1 and dec^2

Gadgets in dec^2 Although gadgets in dec^2 do not need to have printable addresses, it is more complicated and need additional gadgets besides the basic ones. The most notable one is the gadget for conditional jump.

There are a few steps we need to perform in a conditional jump. First, we need to load the next word to be decoded into a register (`eax` in our experiment). This can be done easily with `pop ecx`, `mov edx, [ecx+4]`, and `mov eax, [edx+4]`.

Second, we need to check whether we have reached the end of the encoded shellcode. As discussed in Section 5.2, we use `0x7e7e7e7e` as an indicator. We subtract `0x7e7e7e7e` from `eax`, then `neg eax` to get the corresponding CF flag value. If `eax` is zero, CF will be zero; otherwise, CF will be one. Third, we use CF to help us determine whether we need to add the offset to `addr3` (see Fig 7a). To do this, we set `eax` to `0xffffffff` if CF is zero, or `0x0` otherwise.

Last, we load the offset to `ecx`, and use `and eax, ecx` to set `eax` to either the offset or zero, which is subsequently added to `addr3` and moved to `esp` by using `mov esp, [ecx+0xd8]` to finish stack pivot [30]. The first and the last step can be done easily with the basic gadgets. Fig 7b shows the gadgets used to implement the second and the third steps.

6 Experiments and Discussions

6.1 Experiments

In this section, we perform experiments on our proposed two-layer packer by applying it on two real-world unpacked ROP shellcode. One is a local SEH exploit [20] on Winamp v5.572 originally published at Exploit Database[7]. When users select version history of the vulnerable application Winamp v5.572, a file whatsnew.txt will be read. Due to vulnerabilities in the string reading procedure, an attacker can craft the file whatsnew.txt to overwrite the BOF and triggers SEH. The other is an exploit on RM Downloader v3.1.3[8] which uses the same idea as in the Winamp exploit. Attackers use a crafted media file to trigger SEH in the vulnerable RM Downloader v3.1.3. These two examples both use ROP to

[7] `http://www.exploit-db.com/exploits/14068/`
[8] `http://www.exploit-db.com/exploits/14150/`

call function `VirtualProtect()` to make the stack executable, and then execute the injected non-ROP shellcode to run the calculator (by executing calc.exe).

We download the original ROP exploit and apply our automatic two-layer packer on it. As mentioned in Section 4, our packer generates packed ROP shellcode for both Windows 7 and Windows XP (see Appendix A for the shellcode of the Winamp exploit on Windows 7). Table 4 shows the size of each part in the packed ROP shellcode as well as the number of instructions executed by the original unpacked ROP shellcode and the packed ROP shellcode.

Shellcodes	Aspects	Windows 7		Windows XP	
		Original ROP	Packed ROP	Original ROP	Packed ROP
Winamp v5.572	Printable	No	Yes	No	Yes
	Size of dec^1	N/A	3,316 bytes	N/A	4,216 bytes
	Size of $\mathsf{enc}^1(\mathsf{dec}^2)$	N/A	444 bytes	N/A	676 bytes
	Size of $\mathsf{enc}^2(\mathsf{shell})$	N/A	2,232 bytes	N/A	2,274 bytes
	Total size	1,112 bytes	5,992 bytes	1,132 bytes	7,166 bytes
	# of instructions executed	741	17,325	747	22,932
RM Downloader v3.1.3	Printable	No	Yes	No	Yes
	Size of dec^1	N/A	3,316 bytes	N/A	4,216 bytes
	Size of $\mathsf{enc}^1(\mathsf{dec}^2)$	N/A	444 bytes	N/A	676 bytes
	Size of $\mathsf{enc}^2(\mathsf{shell})$	N/A	42,360 bytes	N/A	42,404 bytes
	Total size	21,176 bytes	46,120 bytes	21,198 bytes	47,296 bytes
	# of instructions executed	21,687	318,285	21,727	324,420

Table 4. Packing shellcode

As shown in Table 4, the sizes of dec^1 and $\mathsf{enc}^1(\mathsf{dec}^2)$ are the same for different ROP shellcode on the same platform. This is because dec^2 uses a loop which has the same size when dealing with different shellcode.

The overhead of the resulting packed shellcode mainly comes from $\mathsf{enc}^2(\mathsf{shell})$. As discussed in Section 5, every 4-byte code in the original ROP shellcode is represented by two 4-byte operands, and therefore the size of $\mathsf{enc}^2(\mathsf{shell})$ is roughly two times of the size of the original shellcode. This is confirmed in Table 4. Note that although polymorphism can be applied and there are many variations in the packed shellcode, they all have the same size.

Also note that the number of instructions executed increases more than 10 times. This is mainly due to the loop in the decoding as discussed in Section 5. Each 4-byte code in the original shellcode needs a few instructions to 1) read the encoded data, 2) calculate and write the decoded word, and 3) conditionally jump to the next round of decoding. However, this increase in the number of instructions executed has small impact on the detectability of the shellcode.

6.2 Discussions and Limitations

We briefly mentioned assumptions we make in decoding and encoding in Section 4 and Section 5. In the rest of this section, we more systematically discuss some issues and potential limitations of our two-layer packer.

64-bit architecture Our two-layer packer works on 64-bit systems, although the probability of a 64-bit address being printable is smaller, i.e., it is more difficult to find printable gadgets. However, the transformation proposed in Section 4 in which non-printable addresses are represented by two or three printable addresses still works on a 64-bit system.

Addresses of data segment We use the data segment as temporary storage for dec^1 and dec^2 and therefore need to know the address of the data segment. This address is not randomized and there are existing tools (e.g., PEreader, readelf) to get it. We could also store temporary data on the stack to eliminate this requirement. However, it will make the design more complex as we need registers to keep the address of the stack. We leave it for our future work.

Base address of libraries When running on Windows XP, our approach works well without additional assumptions since the base addresses of libraries are fixed. However, Windows 7 makes the base addresses of `ntdll.dll` and `kernel32.dll` different after every restart. In our limited tests, we find that the first byte of the base addresses of these two libraries are always printable, and the second byte (random) has roughly 36.7% percent probability of being printable. If these addresses happen to be non-printable, we cannot make use of instructions in these libraries and our packer cannot generate the printable shellcode.

Loading of libraries The libraries we use (`ntdll.dll` and `kernel32.dll`) need to be loaded in the vulnerable application. Since they are common and provide some basic functionality, they are loaded even in the simplest application (whose source code contains only a return statement) generated by normal compilers.

Polymorphism in dec^1 As discussed in Section 4, polymorphism can be obtained in encoding shell and dec^2. However, dec^1 is not encoded and we can only achieve polymorphism in different ways. Most instruction sequences in dec^1 are common instructions. For example, there are 19 useful `pop eax` instructions with different printable gadgets found in `shell32.dll`. We could randomly choose anyone of them. For instruction sequences that are relatively hard to find, we could turn our attention to other equivalent instruction sequences and corresponding gadgets or those with side effects [28]. This is outside the scope of our paper though.

Size of the resulting ROP When the ROP shellcode gets bigger, there might not be enough space on the stack to hold the ROP shellcode. This limits the applicability of our packed shellcode. In addition, some special ROP shellcode gets a value on the stack by using offsets to current `esp`. When the size of the ROP shellcode changes, this offset might be changed accordingly. In this case, we need some manual work to change the offset in the packing.

6.3 Implications

We have demonstrated the idea of packing ROP shellcode and making it printable and polymorphic. We also show the success in applying our packer to existing ROP code. Besides that, our experiment results show that ROP is probably more powerful than what we had believed. Since the introduction of ROP, people have realized its power in circumventing the $W \oplus X$ protection mechanism and started to propose counter-measures to it. In this paper, we show that ROP is also powerful in many other aspects, including being packed, printable, and polymorphic. In other words, ROP inherits many attacking capabilities of existing non-ROP attacks. This has strong implications in further analysis of ROP and its counter-measures.

7 Extensions of our Two-Layer Packer

We have shown the usage of our two-layer packer in making ROP shellcode printable and polymorphic. In this section, we show that our packer can be used in a couple of other scenarios including evading detection by signature-based anti-virus programs and packing shell using ROP without returns.

7.1 AV-immune ROP packer

Although malware based on ROP is not common yet, we expect that anti-virus programs will have more ROP signatures in the near future. Here we investigate if ROP shellcode can be packed to avoid specific signatures to evade detection. The idea is simple. We first scan the original shellcode to find byte streams that match signatures used by anti-virus programs, and then apply enc^2 to use two random 4-byte operands to represent them.

Shellcode produced by our AV-immune packer is similar to the printable shellcode presented earlier, see Fig 8, except that some optimizations can be made when we assume that only a small number of bytes are detected as matching the signatures and only these bytes need to be encoded by enc^2.

Encoding shell works in a similar way except that operands are not printable but random numbers not containing detectable signatures. This leaves many more choices in the operands and consequently better polymorphism.

Encoding dec^2 is also simple. If addresses in dec^1 match a signature, the corresponding gadget cannot be used and we look for alternatives to the gadget. Fortunately, gadgets in dec^1 are quite common and we have multiple choices to select the right gadgets that do not contain a signature.

In order to evaluate our AV-immune packer, we study the signature database of ClamAV 0.96.4 to see if it contains any signatures matching ROP shellcode. We find that the coverage of ROP is extremely small. To better demonstrate the effectiveness of our packer, we randomly choose byte streams in unpacked ROP shellcode and assume that they are used as signatures in anti-virus programs.

Table 5 shows the result when we assume that 5% and 10% of the unpacked shellcode contains signatures used in anti-virus programs. We see that about 170 more bytes are needed in $\mathsf{enc}^2(\mathsf{shell})$ (the last part in Fig 8) in order to encode the additional bytes detected in the original shellcode (with sizes of all other parts remain unchanged). Again, the number of instructions executed increases by a few times, although it has little effect on the detectability (please refer to Appendix B for an example of the packed ROP shellcode).

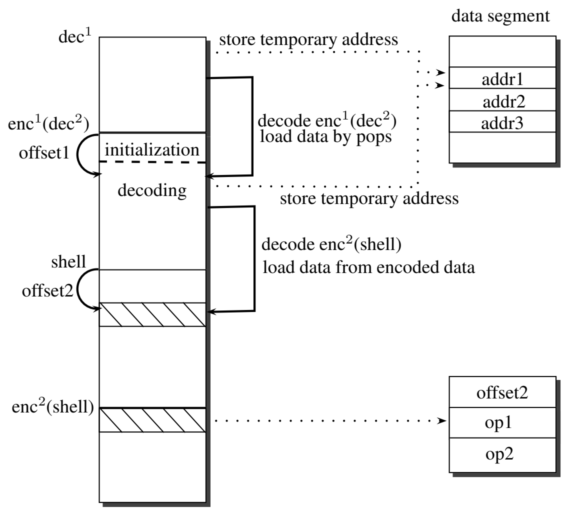


Fig. 8. Anti-Virus immune shellcode

	5% of shell detected		10% of shell detected	
	Original ROP	Packed ROP	Original ROP	Packed ROP
AV-immune	No	Yes	No	Yes
Size of dec^1	N/A	208 bytes	N/A	208 bytes
Size of enc^1(dec^2)	N/A	620 bytes	N/A	620 bytes
Size of enc^2(shell)	N/A	176 bytes	N/A	344 bytes
Total size	1,112 bytes	2,116 bytes	1,112 bytes	2,284 bytes
# of instructions executed	741	2,286	741	3,644

Table 5. Packing Winamp v5.572 ROP shellcode

7.2 Packing shell using ROP without returns

Checkoway et al. proposed ROP without returns [7]. In this section, we show that decoders in printable shellcode produced by our two-layer packer could be constructed without returns. Since useful gadgets without returns are more difficult to find, we extend our search space from common libraries to others including `msctf.dll`, `msvcr90.dll` and `mshtml.dll`. The search space can be further extended to include other binary files when needed.

Table 6 shows some useful gadgets that we find on Windows XP, whose functionality includes Trampoline (an update-load-branch [7] sequence which acts as the `ret` instruction), loading and storing data, and arithmetic.

To make discussions simple, here we assume that the address of the stack is known. As shown in Fig 9, we first use gadget ① to store two operands to `edx` and `edi`, respectively, and then set other registers (e.g., `ecx`, `ebx`, and `eax`) with the corresponding values. Gadget ② acts as a trampoline to jump to the appropriate locations during execution. Gadget ③ carries out calculation using `edx, edi`, and jumps to the trampoline. We next use gadget ④ to store the result of decoding to `eax`. We use gadget ⑤ to load the address of the trampoline to

Purpose	Instruction	Relative address	Library
Trampoline	`add ebx, 0x10; jmp [ebx]`	0x000832f2	msvcr90.dll
	`pop ebx; xlatb; jmp [ebx]`	0x00299637	mshtml.dll
Loading and storing data	`popad; jmp [ecx]`	0x000062af	msctf.dll
	`pop edi; jmp [ecx]`	0x000a2f9f	jscript.dll
	`pop ecx; jmp [edx]`	0x001bd291	mshtml.dll
	`mov [ecx], eax; call edi`	0x0008999f	mshtml.dll
	`mov [eax], edi; call esi`	0x001627fe	shell32.dll
Arithmetic operations	`sub edi, ebx; jmp [edx]`	0x00092b2e	shell32.dll
	`sub ebx, esp; jmp [ecx]`	0x000056e4	mshtml.dll
	`sbb esi, esi, jmp [ebx]`	0x00018fe9	mshtml.dll
	`xor edi, edx, jmp [ebx]`	0x00021e29	mshtml.dll
	`xor edx, edi, jmp [ecx]`	0x00178b2b	ieframe.dll
	`xchg edx, edi, jmp [ecx]`	0x00017e2d	ieframe.dll

Table 6. Gadgets for ROP without returns

`edi`, and gadget ⑥ to load the address of the decoded 4-byte code to `ecx`. In the end, we use gadget ⑦ to finish the final storing operation.

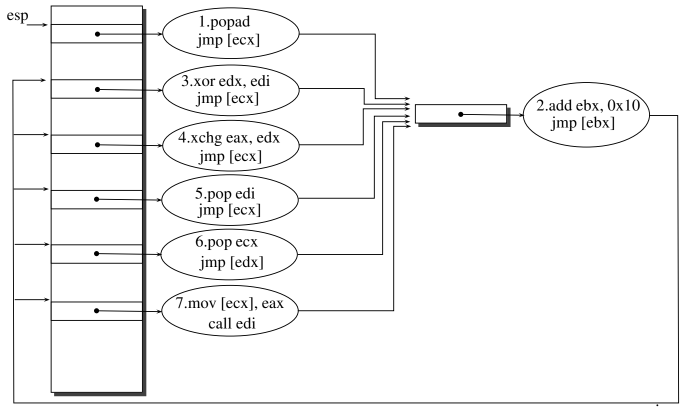


Fig. 9. Arithmetic using non-return gadgets

Table 7 shows the result of our experiment on two ROP shellcode (see Appendix C for the actual shellcode). Note that in our experiment the original shellcode uses ROP with returns. However, our packer is able to pack shellcode that uses ROP without returns, too. We did not perform experiments on that simply because we cannot find existing shellcode that uses ROP without returns.

We have discussed our success in constructing a one-layer packed ROP without returns. As discussed in Section 5, the two-layer packer uses a loop to decode the encoded shellcode which needs a gadget that performs a conditional jump. Unfortunately, after scanning most of the common libraries on Windows XP, we could not find gadgets to perform the appropriate `and` and stack pivot for the conditional jump. We want to stress that this is not a limitation to the idea of our two-layer packer, but simply a limitation of ROP without returns in that

	Exploit example of small size		Exploit example of large size	
	Original ROP	Packed ROP	Original ROP	Packed ROP
Total size	56 bytes	2,072 bytes	1,112 bytes	62,272 bytes
# of instructions executed	19	456	741	17,784

Table 7. One-layer packer without returns

gadgets are more difficult to find. We believe that if we can find the appropriate gadgets in other libraries or programs, our two-layer packer will work.

8 Conclusion

In this paper, we propose a packer for return-oriented programming which outputs printable and polymorphic shellcode. We demonstrate that our packer can be used to pack real-world ROP shellcode to evade signature-based detection and non-ASCII filtering. Extensions of our packer show that the idea of it applies to ROP without returns, too.

Acknowledgments

We would like to thank Professor Puhan Zhang for his guidance of our research and all members of Software Security Research of School of Software and Microelectronics in Peking University for their help.

References

1. W xor X. `http://en.wikipedia.org/wiki/W^X`.
2. K. G. Anagnostakis and E. P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats*, 2009.
3. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security 2003)*, 2003.
4. S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security 2005)*, 2005.
5. Tyler Bletsch, Xuxian Jiang, and Vince W. Freeh. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS11)*, 2011.
6. E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS 2008)*, 2008.
7. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS 2010)*, 2010.

8. S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. In *Proceedings of the 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, 2009.
9. P. Chen, G. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS 2009)*, 2009.
10. L. Davi, A. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2011)*, 2011.
11. T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. V. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack magazine, 9(61)*, Aug. 2003. `http://www.phrack.org/issues.html?issue=61&id=9`.
12. A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS 2008)*, 2008.
13. Aurelien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code (SecuCode09)*, 2009.
14. Greuff. Writing utf-8 compatible shellcodes. *Phrack magazine, 9(62)*, Jul. 2004. `http://www.phrack.org/issues.html?issue=62&id=9`.
15. R. Hund, T. Holz, and F. C. Freiling. Returnoriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security 2009)*, 2009.
16. T. Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-University Bochum, Germany, 2009.
17. J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS 2009)*, 2009.
18. Obscou. Building ia32 'unicode-proof' shellcodes. *Phrack magazine, 11(61)*, Aug. 2003. `http://www.phrack.org/issues.html?issue=61&id=11`.
19. Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarottie, and Engin Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of The 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.
20. M. Pietrek. A crash course on the depths of win32 structured exception handling. *Microsoft Systems Journal*, Jan. 1997. `http://www.microsoft.com/msj/0197/exception/exception.aspx`.
21. Rix. Writing ia32 alphanumeric shellcodes. *Phrack magazine, 15(57)*, Aug. 2001. `http://www.phrack.org/issues.html?issue=57&id=15`.
22. R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications, 2010. Online: `http://cseweb.ucsd.edu/~hovav/dist/rop.pdf`.
23. H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS 2007)*, 2007.
24. H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS 2004)*, 2004.

25. A. Stepan. Improving proactive detection of packed malware. Virus Bulletin, 2006.
26. P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, Feb. 2005.
27. PaX Team. Pax address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`.
28. Z. Wang, R. Cheng, and D. Gao. Revisiting address space randomization. In *Proceedings of the 13th Annual International Conference on Information Security and Cryptology (ICISC 2010)*, 2010.
29. J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, 2003.
30. Dino A. Dai Zovi. Practical return-oriented programming, 2010. `http://trailofbits.com/2010/04/26/practical-return-oriented-programming/`.

A Packed ROP for Winamp exploit on Window 7

dec^1	enc^2 (shell)
6C 62 57 77 41 31 B1 47 78 4F 7B 76 65 65 65 65 56 66 7B 76 65 65 65 65 39 4E 42 42 2E 6F 5E 77 21 21 21 21 2B 63 5F 77 65 65 65 65 34 49 56 77 40 21 21 21 46 3F 7C 76 65 65 65 65 65 65 65 65 65 65 65 65 2A 66 57 77 65 65 65 65 56 66 7B 76 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 73 44 46 28 2E 6F 5E 77 7E 7E 7E 7E 2B 63 5F 77 65 65 65 65 34 49 56 77 40 21 21 21 46 3F 7C 76 65 65 65 65 65 65 65 65 65 65 65 65 48 4A 7C 76	55 3A 53 23 21 41 22 41 55 3E 2A 23 21 41 40 41 6A 4D 31 23 22 22 22 41 74 3E 22 23 22 22 21 41 38 38 38 38 41 40 40 40 32 33 33 33 21 22 22 22 38 38 38 38 40 41 41 41 39 39 39 39 41 40 40 40 39 39 39 39 40 41 41 41 21 6A 2D 23 23 22 22 41 31 30 30 30 22 21 21 21 31 30 30 30 22 21 21 21 31 30 30 30 22 21 21 21 31 30 30 30 22 21 21 21 31 30 30 30 22 21 21 21 31 30 30 30 22 21 21 21 2A 27 5A 23 21 22 22 41 2D 28 24 23 41 40 41 41 ......
enc^1 (dec^2)	dec^2
22 6E 55 77 2E 6F 5E 77 65 65 65 65 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 65 65 65 65 65 65 65 65 22 6E 55 77 2E 6F 5E 77 65 65 65 65 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 65 65 65 65 65 65 65 65 22 6E 55 77 2E 6F 5E 77 65 65 65 65 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 65 65 65 65 65 65 65 65 2E 6F 5E 77 65 65 65 65 56 66 7B 76 65 65 65 65 65 65 65 65 2E 6F 5E 77 65 65 65 65 56 66 7B 76 65 65 65 65	22 6E 55 77 2E 6F 5E 77 81 B0 47 00 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 B3 B0 87 B0 5B 5C 93 76 22 6E 55 77 2E 6F 5E 77 85 B0 47 00 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 B7 B0 87 B0 5B 5C 93 76 22 6E 55 77 2E 6F 5E 77 89 B0 47 00 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 BB B0 87 B0 5B 5C 93 76 2E 6F 5E 77 98 01 00 00 56 66 7B 76 31 B1 47 00 68 F7 5F 77 2E 6F 5E 77 74 01 00 00 56 66 7B 76 35 B1 47 00 ......

B Packed ROP that is av-ammune

dec^1	shell
56 66 7B 76 8B AF BC FF F0 49 54 77 65 65 65 65 A9 C4 53 77 22 6E 55 77 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 A7 50 83 B0 5B 5C 93 76 65 65 65 65 56 66 7B 76 4C FD FF FF F0 49 54 77 65 65 65 65 A9 C4 53 77 56 66 7B 76 DB AE BC FF F0 49 54 77 65 65 65 65 68 F7 5F 77 56 66 7B 76 65 65 65 65 7E 7E 7E 7E 2E 6F 5E 77 5D 2D 3B 7E 2B 63 5F 77 65 65 65 65 48 4A 7C 76 4F 7A 58 77 65 65 65 65 56 66 7B 76 25 3a 36 77 2E 6F 5E 77	74 6C 96 07 1A 10 09 07 3A D8 8D 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 29 13 09 07 67 40 5B 07 65 72 0A 07 67 40 5B 07 65 72 0A 07 29 13 09 07 67 40 5B 07 65 72 0A 07 29 13 09 07 67 40 5B 07 65 72 0A 07 67 40 5B 07 65 72 0A 07 67 40 5B 07 65 72 0A 07 74 6C 96 07 B3 6A 6C 07 A7 41 11 07 74 6C 96 07 1A 10 09 07 3A D8 8D 07 ......
enc^1 (dec^2)	enc^2 (shell)
56 66 7B 76 8B AF BC FF F0 49 54 77 65 65 65 65 A9 C4 53 77 22 6E 55 77 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 A7 50 83 B0 5B 5C 93 76 56 66 7B 76 87 AF BC FF F0 49 54 77 65 65 65 65 A9 C4 53 77 22 6E 55 77 44 59 55 77 65 65 65 65 6C 62 57 77 65 65 65 65 AB 50 83 B0 5B 5C 93 76 56 66 7B 76 83 AF BC FF F0 49 54 77 65 65 65 65 A9 C4 53 77 22 6E 55 77	FF FF FF FF 59 23 68 28 39 73 54 57 FB FF FF FF 73 25 29 2B 6D 77 46 51 FB FF FF FF 43 77 31 26 70 56 22 4B FB FF FF FF 5A 3A 22 23 7E 2A 21 41 FB FF FF FF 31 37 2E 26 53 5E 6C 7C FB FF FF FF 3D 3E 36 38 3F 38 28 34 FB FF FF FF 31 3F 36 2D 52 4F 5D 6B FB FF FF FF 25 3D 33 28 79 48 54 62 FB FF FF FF 26 3B 31 36 7E 45 51 5F FB FF FF FF 25 41 27 2D 2B 74 36 5D ....

C Packed ROP without returns

one round
AF 86 68 74 76 32 51 34 41 41 41 41 41 41 41 41 F2 32 5A 78 67 51 26 6F F2 32 5A 78 F2 32 5A 78 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 2B 8B BC 05 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 31 B5 3D 63 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 9F 2F 42 63 F2 32 5A 78 41 41 41 41 41 41 41 41 41 41 41 41 91 D2 73 63 B4 66 42 01 41 41 41 41 41 41 41 41 41 41 41 41 9F 99 B1 7D 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41