ROPMate: Visually Assisting the Creation of ROP-based Exploits

Marco Angelini*  
Sapienza University of Rome

Graziano Blasilli*  
Sapienza University of Rome

Pietro Borrello†  
Sapienza University of Rome

Emilio Coppa*  
Sapienza University of Rome

Daniele Cono D’Elia*  
Sapienza University of Rome

Serena Ferracci†  
Sapienza University of Rome

Simone Lenti*  
Sapienza University of Rome

Giuseppe Santucci*  
Sapienza University of Rome

ABSTRACT
Exploits based on ROP (Return-Oriented Programming) are increasingly present in advanced attack scenarios. Testing systems for ROP-based attacks can be valuable for improving the security and reliability of software. In this paper, we propose ROPMATE, the first Visual Analytics system specifically designed to assist human red team ROP exploit builders. In contrast, previous ROP tools typically require users to inspect a puzzle of hundreds or thousands of lines of textual information, making it a daunting task. ROPMATE presents builders with a clear interface of well-defined and semantically meaningful gadgets, i.e., fragments of code already present in the binary application that can be chained to form fully-functional exploits. The system supports incrementally building exploits by suggesting gadget candidates filtered according to constraints on preserved registers and accessed memory. Several visual aids are offered to identify suitable gadgets and assemble them into semantically correct chains. We report on a preliminary user study that shows how ROPMATE can assist users in building ROP chains.

Keywords: Malware Analysis, Return-Oriented Programming, Code Reuse, ROP Exploits, Visual Analytics.

1 INTRODUCTION
Code reuse techniques have become prevalent attack vectors against memory vulnerabilities, effectively circumventing traditional system defenses against code injection [13]. Among them, return-oriented programming (ROP) has received considerable attention as it allows an attacker to induce arbitrary behavior in a vulnerable program through a carefully crafted chain of redirections in the program memory, without actually injecting any additional code [19].

A typical attack scenario is based on a controlled stack frame where the return address can be overwritten by means of a buffer overflow. A ROP attack uses short instruction sequences (called gadgets) that are already present in the vulnerable application, as their combination allows for arbitrary computations. Each gadget usually takes the form of a few instructions, with the last one being a return. This allows the attacker to place on the stack a sequence, called ROP chain, made of gadget addresses and immediate operands, that will be executed as a whole thanks to the role of the ret assembly instruction in transferring control between consecutive gadgets.

Building a chain of gadgets is a hard task and this paper aims at progressing in this field by presenting a Visual Analytics system that allows for complementing automatic tools with human intervention – a paradigm successfully explored in other security research [5, 20]. The system provides visual cues that help a human ROP chain builder, making the creation of part or the totality of the chain easier.

Indeed, while existing ROP tools [1, 2] do a very good job in finding useful gadgets, they provide limited support when building complex chains. Recently, solutions have been proposed to build chain portions for carrying out specific tasks only [10]. Overall, no automatic tool currently provides a general solution for dealing with complex dependencies and subtle side effects that often emerge when crafting chains for real-world programs. In this scenario, building ROP exploits remains predominantly a manual task.

The contribution of the paper aims at attacking this problem: it introduces ROPMATE, a Visual Analytics solution supporting the manual construction of the chain for the exploit. The analytical part of the system analyzes the source of the gadgets, producing a list of semantically meaningful gadgets, with the obvious advantage that only gadgets that have a clear effect are maintained and presented to the user. The visual component shows the list of all meaningful gadgets, divided by class and by implemented operation; suitable properties are visually encoded and the user can further filter the list according to her need. Filtering may involve searching for gadgets that implement a particular operation, but, for example, do not modify some registers that have already been set, or access the memory only via some controlled registers, etc.

A preliminary formative user study allowed us to get pros and cons of the proposed approach and led to the development of a revised version of the system.

Summarizing, the contributions of the paper are the following:
• it introduces a novel Visual Analytics environment targeted at exploring and chaining gadgets to produce ROP exploits;
• it explores several analytical and visual solutions that support the user’s task, presenting the most relevant gadgets and allowing for considering alternative solutions, e.g., providing the user with the information of existing gadgets similar to the one s/he is exploring;
• it provides a first feedback about the proposed system, collecting opinions from expert users and analyzing system usage traces to detect similar patterns and improve the system.

The paper is organized as follows: Section 2 presents the scenario in which the system has been developed; Section 3 discusses some related proposals; Section 4 presents the proposed Visual Analytics solution; Section 5 presents a case study; Section 6 presents a preliminary user study; finally, Section 7 draws some conclusions and presents an outlook for future work.

2 APPLICATION DOMAIN

Programs written in type-unsafe languages such as C and C++ are vulnerable to attacks where an adversary corrupts the memory to have execution redirected to an arbitrary code sequence. Buffer overflows are the most frequent form of memory corruptions, with a program input being copied to a buffer without proper bounds checking. In particular, stack-allocated buffers have been historically used by attackers to inject their own code along with legit input data and eventually take control of a program.
Once operating systems designers started incorporating defenses such as Data Execution Prevention (DEP), which hinders code injection by denying code execution from writable regions, attackers devised more subtle and powerful exploitation techniques commonly known as code reuse attacks. Such attacks chain together existing code fragments from the vulnerable application, granting an attacker the same expressive power of a custom injected sequence.\(^1\)

Return-oriented programming (ROP) is the most well-known form of code reuse, and takes its name from the ret assembly instruction that is used to chain together existing code fragments (called gadgets) of the application. ret is commonly used in function epilogues to update the instruction pointer with a value previously stored on the stack to resume execution in the caller function.

Gadgets can be found in libraries, and sometimes within the program itself, using tools that implement variants of the Galileo discovery algorithm [19] such as ROPgadget [1] or ropper [2]. Depending on the size of analyzed code and the maximum length, the number of found gadgets can range from thousands for middle-sized programs, to tens of thousands for large libraries and programs [9]. Gadgets are then analyzed and classified according to the functionalities they provide: in Table 1 we report a possible categorization that we use in the gadget classifier analytical component of ROPMATE.

**Example.** In Figure 1 we present a simple exploit that opens a shell on the machine with the same privileges of the vulnerable application; such an exploit is commonly referred to as shellcode. Before techniques such as DEP were adopted by operating systems, a shellcode consisted of assembly instructions to be injected in the program, as in the left portion of the figure. In particular, in this example a "/bin/sh" string is assembled in CPU register rdi, and then copied to a writable memory area whose address is specified in register rsi. The shellcode then prepares the arguments for the Linux execve system call used to spawn a shell: on a x86-64 architecture, ordinal 0x3b for the call is put in register rdi, and the string in register rsi, and the remaining two arguments – both set to NULL – in rsi and rdx, respectively. Finally, instruction syscall is used to trigger the system call.

An equivalent ROP chain for the shellcode is shown in the middle part of Figure 1. We assume that an attacker placed the chain on the stack so that the stack pointer rsp points to the beginning of the chain when the vulnerable program executes some ret instruction. This instruction updates the instruction pointer rip with the value currently written on the top of the stack, then adjusts the stack pointer before execution continues from the new address in rip. As the stack grows from high to low addresses, rsp is moved to a higher address, i.e., it is incremented by 8 on a 64-bit machine.

In our example, ret loads the address of gadget G1 into the instruction pointer, and the updated stack pointer now points to D1, which contains the address `myGlobBuf` of the buffer where we will

---

\(^1\) Code reuse attack techniques are usually Turing-complete [18].
which, either for demonstration purposes or motivated by criminal
ware. A common belief is that most ROP exploits observed in
when looking for an instruction implementing a particular operation
bugs are discovered in the development of its products, it is important
reasons, are willing to devote a significant effort to building a chain.
manual task [10].
real-world attackers, building ROP chains remains predominantly a
dependencies that may arise as the chain grows, and (iii) limitations on
gadgets that differ only for subtle side effects, (ii) complex depen-
dencies that may arise as the chain grows, and (iii) limitations on
the desired semantics, a ROP chain builder is strongly limited by the
for a traditional shellcode an attacker can choose from the entire
CPU instruction set whichever instruction s/he need to implement
the desired semantics, a ROP chain builder is strongly limited by the
gadgets that are found during the discovery phase [18]. In particular,
when looking for an instruction implementing a particular operation
the following problems may arise:
1. No available gadget contains it. For instance, the chain builder
is looking for a gadget that loads a constant value into a specific
register, say rcx, but no suitable one is found. In the chain of
Figure 1, we had to resort to a gadget G2 that loads a constant
to a different register, specifically rax. Another possibility
is to use multiple gadgets to realize the intended operation:
this happens for instance quite frequently for some arithmetic
operations for which gadgets are notoriously rare [19].
2. A gadget is available, but comes with side effects. This is
the case of gadgets with more than one instruction before the
ending ret: for instance, gadget G1 of Figure 1 not only writes
the content of rax to memory, but also performs a bitwise AND
operation, altering the contents of register rcx. In our example
we do not use rcx to hold any relevant data for later use, thus
this side effect is harmless. However, in general side effects
might clobber (i.e., overwrite) registers holding useful data,
or perform unwanted memory operations that may make the
program crash or ruin the chain.
3. A gadget is available, but is problematic for subsequent opera-
tions. This case is more subtle, as an attacker has not only to
consider the current operation, but also subsequent ones that
will use its results. This happens for instance when some data is
written to some register r. gadgets for subsequent operations
can only read from different registers, and no other gadget can
be used to move data across r and any such register.
In the light of these problems, a ROP chain builder is presented
with a large amount of information originating in: (i) having many
gadgets that differ only for subtle side effects, (ii) complex depend-
dencies that may arise as the chain grows, and (iii) limitations on
choosing a gadget on the basis of the registers that should not be
clobbered at a given point in the chain.
While automatic construction of ROP exploits has been addressed
in previous works, such as the seminal Q paper [18], most available
tools do not work properly in realistic scenarios. Due to the lack of
a publicly available working ROP compiler that meets the needs of
real-world attackers, building ROP chains remains predominantly a
manual task [10].
Applications. In recent years a remarkable number of academic
and security reports have highlighted the power of code reuse
attacks for carrying out complex exploits on mainstream soft-
ware. A common belief is that most ROP exploits observed in
the wild have been written manually by very experienced attackers
which, either for demonstration purposes or motivated by criminal
reasons, are willing to devote a significant effort to building a chain.
From a software house perspective, when a number of memory
bugs are discovered in the development of its products, it is important
to quickly assess whether such bugs are vulnerabilities exploitable
by an attacker, and how dangerous the possible consequences are.
For this reason, a red team made of ethical hackers – either internal
or hired in the market – can evaluate the feasibility of ROP-based
attacks, and to which extent such attacks can cause harm to the sys-
tem. For instance, an attack might take place only under unrealistic
operating conditions, or the attacker has limited freedom in carrying
out certain tasks. It is important to consider that producing a fix for
a vulnerability and validating it before releasing it publicly might
take considerable time. Also, in some cases previous ROP exploits
are adapted when variants of the original vulnerability surface later
on, such as in the case of the EPS component of Microsoft Office.2
Similar considerations might also apply for instance to companies
that have to employ possibly buggy third-party components in, e.g.,
mission-critical systems.
3 RELATED WORK
In cybersecurity, software vulnerabilities are considered one of the
main attack vectors: the Visualization field presents several propos-
als coping with software analysis, ranging from code and structure
analysis [11, 24] to reverse engineering [8], malware analysis [12, 16],
and support for red team activities [25]. Given the specific nature
of the ROP chain building activity, to the best of our knowledge
no Visual Analytics solution has been previously proposed. In the
remainder of this section, we discuss aspects of the ROP literature
that are most related to our ideas.
ROP Defenses. In the arms race between OS designers and
exploit writers, a number of defenses [22] have been proposed during
the last decade to counter code reuse attacks. Address Space Layout
Randomization (ASLR) [15] is one of the first defenses integrated
into modern operating systems: by randomly arranging the address
space positions of key areas of a process, ASLR makes hard for
an attacker to identify gadget addresses. However, ASLR does not
typically randomize the base address of the main executable, leaving
a fruitful source for gadgets. Additionally, ASLR is often defeated
by leveraging a vulnerability that leaks the base address of a library.
Several other defenses can be deployed to limit ROP attacks. For
instance, G-Free [14] rewrites the application code to reduce the
number of available gadgets, while TypeArmor [23] makes gadgets
for function calls unusable outside legitimate control flows. The
main drawback of sophisticated ROP defenses [22] is that they either
come with non-negligible overheads or require heavily customized
compilation toolchains, making them less suitable for production en-
vironments. With respect to ROP mitigations shipped with the latest
releases of Microsoft Windows [3], a recent work [7] shows how
these countermeasures can be bypassed. The work also discusses
how to find and fix realistic expressive gadget sets even in the presence
of advanced ROP defenses.
Automatic ROP Chain Builders. One missing element in the
research landscape is a ROP chain building tool that meets the needs
of real-world attackers, for which building a chain remains mainly a
manual task [10]. Conversely, recent years have witnessed an increase
in the complexity of ROP chains, which moved from being short
sequences aiming at bypassing DEP to enable code injection,
to very complex behaviors encoded entirely in ROP [13]. Gadget
finders, such as ROPgadget [1] and ropper [2], provide very limited
support to a user when building complex chains. While the most
sophisticated tools can attempt to generate chains for a few prede-
fined tasks, such as making the stack executable, they lack flexibility
to support custom actions in an attack, and mostly importantly a
robust methodology for dealing with gadget dependencies and subtle
side effects automatically. As a consequence, they often fail and
generate only partial chains that an attacker must complete manually,
although in some cases such chains turn out to be a dead end.

1https://www.fireeye.com/blog/threat-research/2017/05/
eps-processing-zero-days.html.
4 THE VISUAL ANALYTICS SOLUTION

The proposed solution aims at supporting the user in the whole process from the gadgets extraction to the ROP chain deployment. This section introduces this process, then describes the analytical and visual components that support it.

As first step, the program under inspection is parsed by the analytical component that extracts the set of available gadgets; each gadget is analyzed to identify its semantics and how it interacts with memory and registers. The subset of semantically meaningful gadgets is then loaded in the visual component and the user can start her analysis. In order to build the chain, the user has to iteratively repeat the following steps:

- S/he selects an operation to add to the chain and chooses a gadget with that semantics based on memory and registers constraints (see also Table 1);
- S/he analyzes more deeply the chosen gadget and adds it to the chain if it is appropriate or looks for similar gadgets otherwise;
- When a gadget is added to the chain, the user checks the chain behavior and optionally modifies it by reordering or deleting its gadgets.

When the chain is complete, s/he can export it to a Python script for its binary encoding, a step that is common in the ROP practice.

4.1 The Analytical component

ROPMate relies on different analytics, used in all the steps of the process, from the preprocessing step in which gadgets are characterized, to the management of the developed chain.

Gadgets Semantics: Classification and Filtering. During the preprocessing step, the analytical component analyzes the available gadgets extracted from the program and identifies their semantics using a classification algorithm based on symbolic execution [6]. It groups gadgets that execute the same operation (excluding useless gadgets) and that differ from each other in terms of memory and registers usage. The identified operations are further grouped in classes (see Table 1), thus creating a three-level taxonomy that makes easier to navigate through the different gadgets. The taxonomy allows the user to quickly identify a subset of suitable gadgets; however selecting the right gadget(s) among them require to take into account side effects like the clobbering of registers or unwanted memory operations. For this reason, each gadget representation is enriched with information regarding the reading, modification, or dereferencing of registers and memory operations. This information will be then conveyed through visual means to the user during the chain building process and can be used as further filter mechanisms.

Gadgets Similarity. Once a subset of gadgets that execute the same operation is identified, ROPMate is able to compute a dissimilarity function among them in order to facilitate their exploration. Given a gadget $G_i$, it analyzes its semantics and builds the set $S_i$ containing $G_i$ and all the gadgets that execute the same operation. $S_i$ is then partitioned in $P_i = \{Q_0, ..., Q_m\}$ such that each subset $Q_h$ contains all the gadgets that modify the same set of registers. For every pair $(Q_j, Q_k) \in P_i \times P_i$, the dissimilarity function $d(Q_j, Q_k)$ is computed as follows:

$$d(Q_j, Q_k) = \frac{|R_j \cup R_k - (R_j \cap R_k)|}{|R_j| + |R_k|}$$

with $R_h$ being the set of registers modified by the gadgets of set $Q_h$.
and the relative dissimilarity matrix is built. That allows for quickly locating substitutes of $G_i$ when, for any reasons, $G_i$ cannot be used.

### 4.2 The Visual component

The first step while using ROPMATE is to load the program under examination (see Figure 2.A3); after that it is parsed by the analytical component that extracts and characterizes gadgets as discussed in Section 4.1. Once the list of gadgets is extracted from the program, the user can start to build the ROP chain.

The **Tree Pane** (see Figure 2.C) shows the taxonomy of gadgets, grouped by operations and classes. The taxonomy is represented as a tree allowing to quickly locate suitable gadgets based on the desired operation by descending the tree. The first level of the tree shows the classes (see Figure 3a); a histogram aligned to the list of classes suggests the number of available operations for each class, with the width of the bars proportional to this number. Clicking on a class reveals the list of its operations (see Figure 3b). A second histogram is aligned to the list of operations; the width of the bars is now proportional to the number of available gadgets.

Once the desired operation is identified, the list of corresponding gadgets, partially ordered by ascending complexity (modeled as a heuristic of: number of dereferenced registers, number of modified gadgets, partially ordered by ascending complexity (modeled as a tree allowing to quickly locate suitable gadgets based on the desired operation by descending the tree). The pane contains all the registers, identifiable by their name and color and a checkbox that, when is checked, filters from the search space in order to face the possible side effects: the user is allowed to explicitly define the filter by entering a search text: e.g., “rax register encoded in green”, and have half height otherwise (e.g., rsi and rdi registers encoded in gray and light blue.)

![Figure 3: Details of the Tree Pane showing the first two levels of the taxonomy of gadgets: (a) the first level of the tree shows the list of 11 classes available for the binary and a histogram suggesting the number of operations for each class, (b) the second level of the tree shows the list of 5 operations available for the UnOp class and a histogram to convey information regarding the number of gadgets for each such operation.](image)

**Figure 4:** Details of the 18 available gadgets for the rdi+=1 operation of Figure 3b. Gadgets are identified by their assembly code; the width of the memory requirement bars, attached to the right, is proportional to the binary logarithm of the required memory occupation (the red color of 2 boxes means that the requirement of the gadgets exceeds the memory threshold); the aligned dereferenced registers matrix shows which registers are dereferenced by each gadget, the entries are rectangles with full height if the dependency is not satisfied (e.g., rax register encoded in green), and have half height otherwise (e.g., rsi and rdi registers encoded in gray and light blue.)
the registers that it modifies and the ones that it dereferences; an example is provided in Figure 5. By clicking on the box, the gadget is added to the chain.

While analyzing a gadget, the user might realize that it does not respect certain constraints and thus seeks equivalent gadgets with different side effects. The Similarity Pane (see Figure 2.E and Figure 6 for details) presents all the gadgets that execute the same operation as the selected one using the dissimilarity matrix presented in Section 4.1. MultiDimensional Scaling (MDS) is applied to the matrix to obtain a pair of coordinates for each subset and all the subsets are shown in a scatter-plot, highlighting in green the set containing the selected gadget. Since each point of the scatter-plot represents a set of gadgets, the radius of a point is proportional to the cardinality of the set that it represents. By clicking on a point of the scatter-plot, the relative set is shown in the Tree Pane allowing the user to inspect the contained gadgets in detail.

Adding a gadget to the chain will trigger the recomputation for the registers set so far. By default, the Registers Pane is updated to preserve these registers, thus showing in the Tree Pane only the gadgets that are safe with respect to already set registers. The current state of the built chain is visible in the Chain Pane (see Figure 2.F). The chain is represented as a sequence of rectangles in which each rectangle corresponds to a gadget; the background color of the rectangle is the color of the register set by the gadget. Inside a rectangle, registers used by the gadget are shown by distinguishing between dereferenced registers (on the left of the rectangle) and source registers (on the right). These registers are represented as vertical rectangles using the same convention seen for the dereferenced registers matrix: full height indicates that the dependency is not satisfied at that point of the chain; conversely, half height indicates that it is. This encoding becomes increasingly important as the chain grows to manage complex dependencies between gadgets. Below the rectangle, the operation of the gadget is reported and, for gadgets that set a register to a value, there is a textbox to specify that value (see Figure 7). The sequence can also be changed by reordering gadgets via drag and drop or by removing them.

Once the built chain is complete, clicking on Dump the system will produce a Python script that generates the binary encoding of the chain as shown, for instance, in the right portion of Figure 1.

5 Case Study

In this section we present our case study, showing how ROPMATE has been used to produce a ROP chain for a vulnerable program that has been the subject of the SECCON 2017 capture-the-flag security competition [21]. The application is written in Go: as compiled Go programs ship with the language runtime embedded in the binary, such runtime usually represents a rather rich source of gadgets. However, such gadgets come with subtle side effects that are less frequent for instance in gadgets from compiled C programs. Impersonating a red team member, we will build a ROP chain to spawn a shell from the attacked application, as in the running example from Section 2 used to illustrate the ROP technique.

Security Bug. The application is amenable to a buffer overflow originating in the lack of bounds checking on one of its input parameters. Our goal is to show a way to turn this defect into a security vulnerability that could be exploited by an attacker to execute arbitrary code on the machine. A common practice is to try to build a ROP chain that is able to replace the running process with a shell in the hands of the attacker: we will attempt to execute the system call `execve("/bin/sh", NULL, NULL)` as in Section 2.

ROP Chain Structure. To execute the Linux system call implementing the `execve` functionality, we remind the reader that a chain has to place the corresponding system call ordinal `0x3b` in register `rax`, and the three parameters for the call in registers `rdi`, `rsi`, and `rdx`. Also, as the first argument is a pointer to the string "/bin/sh", the chain has to write the string to some memory region and place its address in the corresponding register `rdi`.

Due to complex register dependencies that might arise, in the ROP practice it is common to first assess whether there are fairly

---

Such competitions propose challenges that reflect real-world problems, but with an artificially higher level of complexity aiming at stressing the abilities of the participants. For instance, binaries may miss some useful ROP gadgets that instead are usually available in real-world programs.
simple gadgets to load values to registers, as this is an operation is prerequisite of most tasks. The builder will then worry about how to make data movements between two registers, or a register and memory. Following this approach, we will build a chain that performs three steps: (i) load constant values to mandatory register destinations for the system call; (ii) write the string "/bin/sh" to memory and have its address available for the call; and (iii) trigger the system call using a gadget containing the syscall instruction.

**Crafting the ROP Chain.** A possible chain implementing the attack can be constructed as follows (the case study can be followed live in the provided supplemental video):

1. We start by temporarily filtering out gadgets that perform memory accesses by using the Avoid Memory checkbox. This will help us to quickly identify gadgets that are unlikely to make the program crash due to possibly invalid memory operations reducing the number of classes in the Tree Pane from 11 to 3.

2. We look into operations from the LoadConst class to determine which registers can be initialized with an immediate read from the stack using a single gadget. For this program, we find such gadgets for registers rax, rbp, rsi, and rcx. By clicking on the rax operation (required to set up the ordinal of the system call), we identify the 6 possible gadgets that implement it and add to the chain the simplest one in terms of memory requirement identifiable by using the memory requirement bars. We then repeat the same workflow for the rsi operation, required to set up the second argument of the system call, and assign the proper input values to the two gadgets (i.e., 0x3b and 0x0, respectively) using the textbox under each newly added gadget in the Chain Pane.

3. The Tree Pane shows that there are no available gadgets for the remaining registers to set up (i.e., rdx and rdi), consequently we are forced to resort to gadgets that access memory by removing the Avoid Memory filter. We move to setting up rdi by typing its name in the Filter Bar and ROPMATE presents us with two alternatives. The dereferenced registers matrix shows us that both of them dereference rax to write to memory, so we choose the first one due to its smaller memory requirement.

4. ROPMATE will highlight for the newly added gadget in the chain a dependency on the contents of rax, which we previously assigned with the 0x3b value. However, rax at this point should contain an address such that the memory write performed by the gadget takes place in a safely writable region. To this aim, we change the 0x3b value for the first gadget with an identifier valid_address that will eventually be assigned in the Python script. Finally, we write the constant value that will be loaded to rdi: we choose another writable-address identifier bin\_sh\_address.

5. We then proceed by “restoring” the system call ordinal in rax: we copy the gadget that loads a constant to it by clicking on the Analysis Pane; the gadget is duplicated and added at the end of the chain. We then set its value to 0x3b.

6. For loading register rdx with the third argument of the call, we follow a similar workflow as for rdi. Using the Filter Bar, 10 gadgets implementing the required LoadConst operation can be found: the combination of memory requirement bars and dereference registers matrix leads us to choose one of the first 3 gadgets that will perform a memory write controlled by rax, which we can make happen within the same safe region as for the gadget that sets rdi. The required 0x0 value is assigned to the operation, then the gadget is dragged with the mouse right before the one that loads 0x3b to rax, so that valid_address will be used for the memory access. At this stage, step (i) has been carried out by the chain.

7. We now need to make provisions to assemble "/bin/sh" in the memory location bin\_sh\_address pointed by rdi. Using the Filter Bar, we type *rdi to highlight the relevant gadgets that write to the pointed location. We choose the first operation among the three available as it conveniently reads the value to write from rax. Among the 6 possible gadgets for it we choose the one that dereferences only rdi as suggested by the dereferenced registers matrix. Similarly as when assigning rdi, we add a duplicate instance of the gadget that loads to rax at the end of the chain, and then we modify the second-last instance of such gadget to hold the required string, which is encoded along with its terminator using the handy Python function u64("/bin/sh\_x000020") . Step (ii) is thus completed.

8. Step (iii) is straightforward: we look for a gadget containing a syscall instruction by looking within the Other class in the Tree Pane, and find a gadget that implements the intended semantics without any side effects. The ROP chain is thus complete, and can be exported to a Python script using the Dump functionality.

6 User evaluation

We have conducted a formative user evaluation in order to obtain some preliminary feedback on the general usability and effectiveness of ROPMATE. The involved subjects were 4 ethical hackers having experience in red teaming processes, ROP exploits, and hacking competitions for more than one year.

Participants were first exposed to a practical use of ROPMATE and, after that, they were asked to accomplish some tasks on the system and to express their thoughts, feelings, and opinions while interacting with it. In order to further improve the obtained feedbacks, we have used an evaluation environment [4] that is able to encapsulate ROPMATE and trace the participants’ actions.

Concerning the received feedbacks, all participants reported the visual approach to the problem was very useful and user-friendly. In particular, P2 and P3 observed that ROPMATE acted as an interactive recommender for the next gadget to add to the chain. Participant P4, instead, commented that the proposed way to create the ROP chain allowed him to not be overwhelmed by useless gadgets. Furthermore, all participants thought that the interaction technique of clicking and filtering gadgets in the Tree Pane was easy to understand and found the classification helpful for searching efficiently the right gadget. Participant P1 and P2 appreciated that the histograms near the classes allowed them to have a visual hint of the number of different operations among which they can choose. Participant P3 expressed some frustration about the dragging of gadgets in the presence of a long chain, and participant P4 commented that it would have been nice to support advanced filters.

This preliminary feedback allowed us to improve the system usability by modifying the visual component of the first prototype: for example, a participant observed that the information of how many gadgets were present in each subset of MDS could be useful for getting an overview of register modifications for a single operation and therefore we added it to the system. The first prototype represented the registers dereferenced by a gadget as a list of colored rectangles. After the evaluation, we changed the list into the matrix representation because some participants asked for a better representation of them in order to better distinguish different registers. Concerning the current state of the built chain, the registers dependency of each gadget was added to the prototype after some participants highlighted the usefulness of this information, specifying also to encode whether the dependency is satisfied or not.

Moreover, the analysis of traces pointed out that users were frequently inspecting the actual size of the gadgets (likely to understand whether they will fit the available stack room for the chain) so we...
have added a user configurable threshold to the system triggering a color scale that allows quickly identifying gadgets below the threshold. As a comparative example with respect to Figure 2, Figure 8 shows the ROPMATE interface before the user evaluation.

7 Conclusion

Building a ROP chain is not easy: understanding the semantics of each gadget and finding the right one amidst data dependencies and side effects can be a heavy burden in the ROP coding activity.

In this paper we have proposed ROPMATE, a novel Visual Analytics solution specifically designed to assist human red team ROP builders providing a visual interface based on semantically meaningful gadgets that can be filtered according to user needs and chained to form fully-functional exploits. This approach has been demonstrated through an example of usage and a formative evaluation.

In the following, we discuss some limitations of the proposed solution, limitations that will constitute the basis for future work.

Filtering Effectiveness. Although the system include means for filtering the gadgets, it does not support complex filtering activities, being the underlying query language somehow limited.

Gadget Chain Overview. User experience suggests that, the longer the size of the chain, the harder the task of assembling it. As an aid for the construction of long chains, we are currently designing an Overview Pane that allows visualizing the whole chain by preserving its main features while inspecting, on demand, details on gadget subsequences.

Guidance. An interesting future work would be to increase the analytical component in order to suggest some combinations of gadgets that perform a specific operation (possibly selected through a more powerful filtering language). This would allow the user to emulate the semantics of a single missing gadget, with the combination of a few others. Such combination of gadgets would then be added as a whole to the ROP chain. However, implementing such a guidance is not an easy task due to the inherently combinatorial nature of the problem. We regard Progressive Visual Analytics [17] as a promising technique that can provide approximate early results in this context.

References