# URSID: Automatically Refining a Single Attack Scenario into Multiple Cyber Range Architectures

Pierre-Victor Besson, Valérie Viet Triem Tong, Gilles Guette, Guillaume Piolle, Erwann Abgrall

# URSID: Automatically Refining a Single Attack Scenario into Multiple Cyber Range Architectures

Pierre-Victor Besson[1,3⋆], Valérie Viet Triem Tong[1,2], Gilles Guette[1,3], Guillaume Piolle[4], and Erwan Abgrall[2]

[1] Inria, Rennes, France
[2] CentraleSupelec, Rennes, France
[3] University of Rennes, France
[4] Thales, Rennes, France

**Abstract.** Contrary to intuition, insecure computer network architectures are valuable assets in IT security. Indeed, such architectures (referred to as cyber-ranges) are commonly used to train red teams and test security solutions, in particular ones related to supervision security. Unfortunately, the design and deployment of these cyber-ranges is costly, as they require designing an attack scenario from scratch and then implementing it in an architecture on a case-by-case basis, through manual choices of machines/users, OS versions, available services and configuration choices. This article presents URSID, a framework for automatic deployment of cyber-ranges based on the formal description of attack scenarios. The scenario is described at the technical attack level according to the MITRE nomenclature, refined into several variations (instances) at the procedural level and then deployed in virtual multiple architectures. URSID thus automates costly manual tasks and allows to have several instances of the same scenario on architectures with different OS, software or account configurations. URSID has been successfully tested in an academic cyber attack and defense training exercise.

**Keywords:** Network security · Computer security · Attack scenario · Cyber Range · Cyber Security Education.

## 1 Introduction

The numbers of cybersecurity incidents keeps increasing every year and cybercrime is nowadays a threat to any organization. In 2021 the FBI reported a 64% increase in losses related to cybercrime compared to 2020, and more than 4 times what it was in 2017 [6]. In particular, the rise of Advanced Persistent Threats (APT) has proven problematic for companies and governments alike, leading to massive data breaches, spying and ransomware-based extortion campaigns.

⋆ Pierre-Victor Besson is funded by the Direction Générale de l'Armement (CREACH LABS)

Advanced Persistent Threats are well organized stealthy threat actors, who gain unauthorized access to an information system and for extended periods of time in order to avoid detection and better reach their goals. In order to deal with this increasing threat, cyber defenders have an array of tools at their disposal, one of them being cyber-ranges.

Cyber-Ranges are defined by the National Institute of Standards and Technology (NIST) [15] as "interactive, simulated representations of an organization's local network, system, tools, and applications that are connected to a simulated Internet level environment". They are a complete information system similar to those used in production, on which a previously specified attack scenario can be run. A cyber-range can be used to improve the technical level of the security teams. A so-called red team attacks the cyber-range. Their goal is to reach the target as soon as possible while remaining undetected. Such a training allows the red team to acquire the reflexes and working methods of the attackers and an efficient threat hunting during a real incident response. In order to guarantee an entry point and make the exercise valuable, cyber-ranges may be populated with flawed configurations and exploits on purpose [5,18,21,22]. A so-called blue team defends the cyber-range. Its objective is to track down the attacker in the cyber-range, to prevent his progression, to explain the meaning of the security alerts and thus ultimately to improve the supervision tools.

The deployment of a cyber range first requires the definition of an attack scenario and an architecture in which this scenario can effectively be played. The scenario definition includes the initial compromise of the system by the attacker, an exit point that represents the goal of the attacker (this can range from user accounts to be compromised, database to be exfiltrated, services to be rendered inaccessible, *etc.*), possibly one or more milestones. The architecture must contain at least the machines, user accounts, services, files and configurations to play the scenario. This architecture must also be populated with credible data, system and network activities similar to what would exist on a real architecture in use. A cyber-range tends to be specific to a unique chain of exploits, and its implementation requires very specific combinations of machines, accounts, software , operating system or configuration files. Once such a cyber-range has been used by teams or tools, it is rendered of little value because both the attacker and the defender know the scenario. Another one must then be created, which comes with additional design and deployment costs.

In this article, we present URSID an automated cyber-range deployment pipeline. URSID automatically deploys several architectures where multiple variations of a same attack scenario can be run and aims to address the challenges raised by cyber-range reusability and description genericity. Our contribution can be summarized as in the following perspectives:

- **High level attack scenario definition using a graph-based model** (Section 3). We propose a model to describe both an attack scenario and the attacker's required skills to achieve this scenario. The description first occurs at the technical level according to the MITRE nomenclature. Similarly to a

kill chain model, it gives a first general overview of the different movements of the attacker in the targeted architecture.
- **Instantiation through multiple low-level attack scenarios** (Section 4.1). We provide a methodology to refine an high level scenario towards more precise descriptions. This refinement process provides variations at the procedural level of the initial scenario in a way that guarantees the consistency of the resulting architecture, by associating attack procedures with architecture constraints.
- **Available implementation**. We provide a git repository of URSID [2] showcasing the entire generation process, alongside an attack scenario which was used as part of a live experiment and instructions on how to refine and deploy it.

## 2 Background

The generation of a cyber-range first requires choosing an attack scenario and then instantiating it on an actual architecture. In our opinion, this need lies at the junction of two types of work in the literature. On the one hand, models that represent architectures that have been or are likely to be attacked. On the other hand, the models that represent the attacker's skills.

*Attack graphs* Attack graphs are formal structures aiming to represent one or more possible attacks on an architecture using nodes and transitions between those nodes. They differ in the literature by how they decide to represent the architecture (which depends on their use case). In particular, host-based models (such as in [1] are attack graph models in which nodes represent an architecture device and edges the access available between these devices. This type of model seemed the most relevant for our purposes, as information about devices can be more easily converted into virtual machine configurations than system states or a list of exploits. Mensah [12] proposes an attack graph model with a novel approach on host-based graphs. Nodes hold information about not only the device the attacker is currently logged on, but also their level of privilege on this connection. A node may also indicate whether it contains interesting data for the attacker, such as a password or a secret file. An attacker may move in this model by taking a transition between 2 nodes, which may have preconditions or consequences on the state of the devices and the attacker. Some transitions have triggers, which indicate that a transition can only be taken if an other specific node has been visited by the attacker beforehand. The formalization used by URSID shares similarities with this work such as the choice of representing nodes as devices and users but differs significantly as a consequence of different use cases. Indeed, Mensah's proposed formalization was ultimately designed for modeling existing architectures, a goal opposite to our own goal of creating architectures from scratch. In summary, their work thus contains a level of exhaustivity in particular in how modeling a transition requires specifying all consequences and conditions of that transition which would make it difficult to

a) write scenarios manually and b) deploy different scenarios based on a single description. Furthermore the set of actions available to the attacker used in [12] to label their transition is not clearly defined, putting actions such as "CVE-2009-1918" or "CONNECT" on the same level. A clear set of attacker actions is required for our purposes in order to properly convert scenario descriptions into architectures.

*Attacker representation* The modeling of attacker actions depends on the scientific objective. Based on observations of actual attacks, the MITRE corporation proposes the ATT&CK Matrix [13], a cyber-adversary behavior knowledge base aiming to describe the different steps of a given attack scenario. In this article, we rely on this nomenclature to describe attacker actions, which are defined on 3 levels of abstraction known as Tactics, Techniques and Procedures, or TTPs. Tactics represent the main goal of an action performed by an attacker an attacker may try to gain higher-level permissions on a system or network (TA004: *Privilege Escalation*).Techniques represent the means used by an attacker to reach his tactical goals: an attacker may exploit a vulnerability to elevate his privileges on a machine (T1068: *Exploitation for Privilege Escalation* [5]). Finally, procedures describe the specific implementation details of an attack: an attacker may attempt to exploit a flaw in the pkexec process on older Linux versions [16] in order to escalate his privileges on a machine. The ATT&CK matrix is currently comprised of 14 tactics, over 200 techniques and describes a few procedures for each technique. There is however to our knowledge no exhaustive list of procedures for a given technique, although some projects (such as [9]) provide tools to make such a connection. Reusing this nomenclature provides URSID with a standardized way to describe any given step of a scenario, which is needed for its application. The ATT&CK matrix is also used by several other projects in the literature [9] [20] [17] or by attack reports and emulation compendiums [14]. While these projects ultimately have different scopes and goals than URSID, reusing the ATTA&CK matrix makes any potential future combination of these tools easier.

## 3 Attack Scenario at the technical level

### 3.1 Representation of an attack scenario

In this work, we propose to describe an attack scenario through a directed graph of attack positions. We model the attacker through the footholds he acquires by holding attack positions. An attacker may progress from one attack position he holds to another if he is able to execute the attack corresponding to the transition between these two attack positions. An attacker can also remain in the same attack position but increase his knowledge by discovering information hosted on the compromised account/machine. Its progression is made possible by the use of an attack technique, itself implemented by the execution of an attack

[5] We are here using the official MITRE numerotation for techniques

procedure. Depending on the required level of abstraction, a scenario may be described on a technical or on a procedural level. This section describes attack scenarios at the technical level. The proposed formalism is generic. It can be used to describe one or many attack paths allowing to reach a targeted attack position. This formalism does not require the description of all the elements of the architecture but only the useful part of the scenario.

An **attack position** corresponds to a session of an attacker, under the identity of a user on a machine of the compromised network. For a given architecture we define $\mathcal{M}$ as the set of all machines available in the architecture, $\mathcal{U}$ as the set of all users with an account on these machines and $\mathcal{D}$ as the data existing in the architecture. The data can be credentials, passwords, cryptographic keys, addresses, or any other data useful to the attacker to progress through the scenario. We define an attack position by a pair $(m, u)$ where $m \in \mathcal{M}$ is a machine of the compromised network and $u \in \mathcal{U}$ a declared user.

Among these attack positions, a **starting position** is a position controlled by default by the attacker at the beginning of the scenario and from which he will be able to play the scenario. The starting position is usually $(Attacker, SuperUser)$, we consider that the attacker has full control over his own machine(s), but could also be a machine from the network, in the case of an insider threat. A **winning position** is a position that is of particular interest to the attacker and which corresponds to one of his ultimate goals in the architecture. For instance this may correspond to an account hosting sensitive data or the admin account of a Windows Active Directory domain controller. In this formalization, we consider that an attacker can hold multiple attack positions at the same time, by compromising multiple accounts on several machines concurrently.

The progression of an attacker in a compromised network is made possible by the use of an attack technique, itself executed through an attack procedure according to the terminology proposed by the MITRE. In our model, this is formalized by edges between the nodes (attack positions) of a scenario graph.

Finally, an attack scenario at the technical level and over an architecture with $\mathcal{M}$ machines and $\mathcal{U}$ declared users is a graph $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$ where

- the set of nodes $\mathbf{P} \subseteq \mathcal{M} \times \mathcal{U}$ is a set of attack positions.
- the set of edges $\mathbf{A}^{\text{tech}}$ is $(p_1, p_2, (\tau, \mathbf{pre}, \mathbf{post}))$ where:
  - $p_1, p_2 \in \mathbf{P}$ are attack positions,
  - $\tau$ is an attack technique (or an attack procedure depending on the level of abstraction),
  - $\mathbf{pre} \in \mathcal{D}$ are the data required to execute $\tau$
  - $\mathbf{post} \in \mathcal{D}$ are the data granted to the attacker when he successfully completes $\tau$

Note that while URSID shares similarities with the modelization of Mensah [12] - particularly in how nodes are defined -, the use of secret requirements and rewards as well as the ATTK&CK matrix are both unique to this work.

For a given scenario $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$, we model the attacker's progression as **an attack state** defined by $\mathcal{A} = (\mathcal{P}, \mathcal{K})$, where $\mathcal{P} \subseteq \mathbf{P}$ is the set of all attack positions he controls, and $\mathcal{K} \subseteq \mathcal{D}$ the data he acquired. In our model, we

consider that an attacker may not lose the control of an attack position after he acquires it once. Indeed, we suppose that once he manages to open a session on a given machine by taking a sequence of transitions, the attacker is able to get back to that attack position by applying the same sequence, or by using a persistence tactic [13]. While this hypothesis may exclude some destructive attacks or unstable exploits, these represent a small proportion of attacks in practice and considering node acquisition to be permanent will make it easier to traverse the graph in later algorithmic treatments.

The attacker progresses in the scenario by controlling new attack positions or increasing his knowledge on the architecture. If an attacker controls an attack position $p_1$ and if in the attack scenario there exists a position $p_2$, a technique $\tau$, two data sets **pre** and **post** such that the attacker already knows **pre** then the attacker can progress from $p_1$ to $p_2$ by applying $\tau$. If the execution of $\tau$ succeeds, the attacker is rewarded with **post**.

An attack path is thus a finite sequence of attack states $\mathcal{A}_0 \xrightarrow{\tau_1} \mathcal{A}_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_n} \mathcal{A}_n$. An attacker can progress from a state $\mathcal{A}_i = (\mathcal{P}_i, \mathcal{K}_i)$ to a state $\mathcal{A}_{i+1} = (\mathcal{P}_{i+1}, \mathcal{K}_{i+1})$ by applying a technique $\tau$ in a scenario $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$ if the attacker controls the attack position $p_i \in \mathcal{P}_i$ and there is a transition to move from $p_1$ to the position $p_2$ i.e. $(p_1, p_2, (\tau, \mathbf{pre}, \mathbf{post})) \in \mathbf{A}^{\text{tech}}$ and the attacker masters the prerequisite to apply the transition $\mathbf{pre} \in \mathcal{K}_i$. Then the attacker can progress to $p_2$: $\mathcal{P}_{i+1} = \mathcal{P}_i \cup \{p_2\}$ and its knowledge increases $\mathcal{K}i + 1 = \mathcal{K}_i \cup \mathbf{post}$. An attack path $\mathcal{A}_0 \xrightarrow{\tau_1} \mathcal{A}_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_n} \mathcal{A}_n$ is said executable if the attacker can progress from $\mathcal{A}_0$ to $\mathcal{A}_n$.

Among the scenarios which can be described with this formalism, we distinguish the so-called **winnable scenarios** in which there is, at least, one executable attack path $\mathcal{A}_0 \xrightarrow{\tau_1} \mathcal{A}_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_n} \mathcal{A}_n$, a starting attack position $p_0$ such that $p_0 \in \mathcal{P}_0$ and a winning position $p_w$ such that $p_w \in \mathcal{P}_n$.

### 3.2 Smash and Grab attack case study

To illustrate this article, we rely on the MITRE Adversary Emulation Plans [14], which outlines the behavior of persistent threat groups mapped to ATT&CK techniques. Our first case study scenario is directly inspired from the "smash and grab attack" of Day 1 of the MITRE APT29 Adversary Emulation Plan and is referred in the following as "Smash and Grab attack scenario". In this scenario, the attacker first gets access to a machine and escalates his privileges on it. He then acquires passwords and files from this position before performing lateral movement to an other machine, which he will also loot and exfiltrate files from.

Our own scenario thus follows the same outline, with a few differences. First, some techniques (such as ones related to tactics like Extraction or Defense Evasion) do not usually lead to a new position or secrets for the attacker, and are less relevant to our generation purposes. Thus, while it would be possible to include these techniques anyway using the same formalism , most of them were left out in order not to clutter the scenario. We also added 2 more machines in

order to give the attacks more options after lateral movement and to diversify his possible attack paths.

The Smash and Grab attack scenario is played out on 4 machines, named *Bear, Raccoon, Badger and Skunk*. Each machine has 2 users, with one of them being a $SuperUser$ with elevated privileges. There is also an additional node $(Attacker, SuperUser)$[6] representing the starting position of the attacker. The goal of the attacker is to reach the winning position $(Skunk, root)$.

At a technical level, the Smash and Grab attack scenario can be described by the graph $\mathbf{S}_e^{\text{tech}} = (\mathbf{P}_e, \mathbf{A}_e^{\text{tech}})$ where $\mathbf{P}_e = (p_i)_{i\in[0,8]}$, $\mathbf{A}_e^{\text{tech}} = (\tau_i)_{i\in[0,15]}$, $\mathbf{S}_e = \{p_0\}$ and $\mathbf{W}_e = \{p_8\}$. A graphical representation of this scenario detailing the values of $(p_i)_{i\in[0,8]}$ and $(\tau_i)_{i\in[0,15]}$ is available in Figure 1 where the notation `Txxxx` refers to the official MITRE numerotation for techniques. The attacker has 2 main paths from the starting position to the winning one: one which requires him to access machine *Raccoon* to acquire credentials in order to perform T1078 on machine *Skunk*, and one which performs T1053 instead. It has to be noted that machine *Badger* is unnecessary for both of these winning attack paths here, acting as a dead end to diversify the paths an attacker may explore.

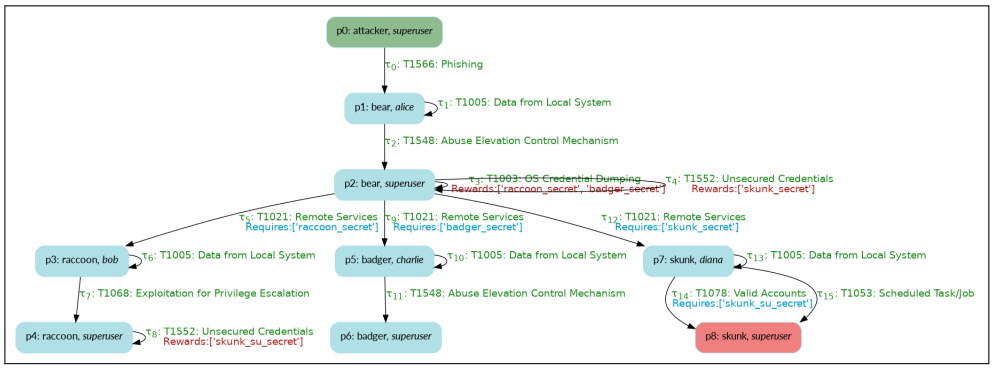


Fig. 1: Scenario "Smash and Grab attack" at a technical level

## 4 Scenario instances at the procedural level

### 4.1 From attack techniques to attack procedures

The high-level description of a technical-level attack scenario is by design not sufficient to grasp the implementation details necessary to actually deploy the cyber-range. The cyber-range architecture is defined by its machines (including

[6] We throughout this work will refer to the attacker's machine as Attacker and consider they have complete control of it.

the version of the operating system, the version of the installed services, the files and their contents) and the declared users (names, passwords and privileges) on these machines. To reach this level of detail we propose to refine the description of an attack scenario at the level of attack procedures.

For each technique $\tau$ allowing to move from an attack position $(u_1, m_1)$ to a position $(u_2, m_2)$, the choice of an attack procedure $\pi$ instantiating $\tau$ induces architectural constraints on machines $m_1$ and $m_2$. We consider here that an attack procedure **instantiates** an attack technique when it is a possible practical implementation of this technique.

The choice of one procedure to instantiate a technique implies constraints on the machines of the architecture on which the scenario will be played. A **constraint** $\mathcal{C}$ is related to a specific machine $m$ and is denoted by $\mathcal{C}(m)$. A single constraint $\mathcal{C}$ includes Operating System constraints, Account constraints, Software constraints and File constraints. Formally, we define $\mathcal{C}(m) = \{OS, Account, Software, Files\}$, with $OS = \{type, version\}$, $Account$ a list of $\{name, group, privilege, services, credentials\}$, $Software$ a list of $\{software, version, port\}$ and $Files$ a list of $\{path, permissions, content\}$. Note that some procedures (such as ones related to techniques corresponding to the Extraction tactic) do not induce any architectural constraint: the attacker is free to use their own tools and means to execute them, regardless of what is available on the machine.

In addition to its architectural constraints, a $\pi$ procedure may have $\pi_{pre}$ preconditions, i.e. external knowledge required by the attacker to execute the procedure successfully. Similarly, the execution of a procedure may provide the attacker with additional knowledge we refer to as scenario secrets, which he needs to discover to advance. These secrets can be identifiers, IP addresses, machine names, the contents of certain files and so on. Each procedure therefore has its own list of required and rewarded secrets that corresponds to the **pre** and **post** defined at the technical level. $\pi_{secrets} = \{\textbf{pre} : [type_1, \ldots]; \textbf{post} : [type_1, \ldots]\}$

A difficulty score is also associated with each procedure to approximate difficulty level to each generated scenario by summing the scores of every procedure involved. This difficulty score, chosen by the scenario designer as they implement procedures, may be tweaked depending on the skill level of the red teams which will attack the scenario. This may provide additional data to the defenders, for instance by studying the correlation between an instance's difficulty and the speed at which it gets compromised, or to provide attackers with a greater variety of challenges in case their skills are heterogeneous.

### 4.2 Inferring cyber-range configurations

A procedural level scenario follows the same formalism as a technical level scenario, except that the edge sets are labeled by attack procedures rather than by techniques. The constraints generated by the choice of procedures make it possible to specify the architecture when these constraints are not inconsistent between them. We propose a **backtracking refinement algorithm** (Algorithm 1) to generate an architecture that can play a scenario described at the

procedure level. This algorithm walks through the scenario graph and try to replace each attack technique labeling an edge in the scenario by an attack procedure instantiating the technique as long as the constraints on the architecture stay coherent. The rules for adding constraints are given in Figure 2 . Intuitively, two constraints on the same machine add up if they have different software, users, credentials or if one of the two constraints is more restrictive than the other in terms of OS and software versions. Otherwise, they are considered to be inconsistent.

To this aim, we need to determine whether individual subconstraints (OS, Accounts, Software and Files) on comparable parameters are compatible, and if so, the result of their combination. We therefore introduce the $\bowtie$ operator, applicable to the various types of subconstraints already defined. Given two subconstraints $Sc_1$ and $Sc_2$, $Sc_1 \bowtie Sc_2$ returns a new subconstraint $Sc_3$, which can be either $\perp$, which is the unsatisfiable subconstraint and a conclusion of incompatibility, or a valid combination of $Sc_1$ and $Sc_2$. Of course, the details of this combination depend on the nature of the operands. Formally, there is a different $\bowtie$ operator for each type of subconstraint. For the sake of clarity, we choose to use the same symbol, while always making sure that both operands are of the same type. Let us take OS constraints as an example. An OS constraint is a tuple of a type subconstraint and a version subconstraint. Quite intuitively, the $\bowtie$ operator applied to "type = Linux" and "type = Debian" would return "type = Debian", while on "type = Linux" and "type = Windows" it would return $\perp$. This version of the operator cannot be defined in a very formal way. When it comes to version numbers, however, it behaves much like a linear constraint solver, under the hypothesis that version numbers can always be compared in a safe manner. In a comparable fashion, we define the semantics of the $\bowtie$ operator on other types of subconstraints. Based on this combination operator, we are then able to define a fusion operator $\sqcup$ over constraint sets, and to determine whether the result remains satisfiable or not. Given two constraints $C_1$ and $C_2$, $C_1 \sqcup C_2$ returns a new constraint $C_3$, which can be either the unsatisfiable constraint $\perp$ or a valid combination of $C_1$ and $C_2$.

Finally, the set of all constraints over the resulting architecture is represented as a dictionary of constraints $C$ containing entries for every machine $m$. This dictionary will be filled and edited throughout the course of our backtracking algorithm, and is considered to be unsatisfiable (C = $\perp$) if any of its entries are unsatisfiable. It is to be noted that any refined scenario (described on a procedural level) will be executable/winnable as long as the scenario described on a technical level was itself executable/winnable. Indeed any winning attack path available to the attacker on a technical level can also be followed on a procedural level by chaining the procedures corresponding to the techniques executed in this attack path.

The virtual machine configuration file resulting from all the procedure choices and their associated constraints is a list of machines and their configuration. For each machine $m$, the corresponding entry in the configuration file $Conf[m]$ is initialized with generic values and constantly updated as constraints are added

| **Subconstraint type** | $Sc_i^{type}$ | $Sc_1^{type} \sqcup Sc_2^{type}$ |
|---|---|---|
| OS | $type_i, version_i$ | $((type_1 \bowtie type_2), (version_1 \bowtie version_2))$ |
| Account | $name_i, group_i, priv_i, serv_i, cred_i$ | $(name_1, group_1, priv_1, serv_1 cred_1) \sqcup (name_2, group_2, priv_2, serv_2, cred_2)$ if $name_1 \neq name_2$<br>$(name_1, (group_1 \bowtie group_2), (priv_1 \bowtie priv_2), (serv_1 \bowtie serv_2), (cred_1 \bowtie cred_2))$ if $name_1 = name_2$ |
| Software | $type_i, version_i, port_i$ | $(type_1, version_1, port_1) \sqcup (type_2, version_2, port_2)$ if $(type_1 \neq type_2 \wedge port_1 \neq port_2)$<br>$(type_1, (version_1 \bowtie version_2), (port_1 \bowtie port_2))$ if $type_1 = type_2$ |
| Files | $path_i, perm_i, content_i$ | $(path_1, (perm_1 \bowtie perm_2), (content_1 \bowtie content_2))$ if $path_1 \neq path_2$<br>$(path_1, perm_1, content_1) \sqcup (path_2, perm_2, content_2)$ if $path_1 = path_2$ |

Fig. 2: Constraint combination rules for 2 procedures

through Algorithm 1. Note that these initial constraints are customizable: for instance one may wish to restrict machines to a specific set of OS, or to install logging software by default. This configuration file contains enough information about the machines comprising the network - which users to add, which software and OS to install, which files to add and create - to be used for virtual machine deployment, a process we will describe in Section 5. We also make sure that the set of transitions $\mathbf{A}^{\text{tech}}$ is sorted to put transitions requiring or reward secrets first, in order to optimize computation times. This is done because secret types are one of the more common causes of architecture incompatibilities. Secrets throughout the scenario refinement process are generated on a first come first served basis. Each time a procedure is picked that requires or rewards secrets, a value for this secret will be generated according to its type (such as ssh key or plaintext password) if it has not been generated already. After the refinement process, some values (such as account credentials or privilege) may have no constraints attached to them, in which case they are set to default values.

We implemented at least 2 procedures for each technique involved in scenario 1, some of which are detailed with their constraints in Figure 3. We then ran the backtracking refinement algorithm on the scenario. This process goes through every transition (starting with the ones requiring or rewarding secrets), picks one of the available procedures for that transition at random and checks for architectural constraints incompatibility. This experiment resulted in a variety of different architectures, all corresponding to the same scenario that was described on a technical level, but having a variety of operating systems, file corpuses, software, services and randomized passwords. Some remarks on the process are as follows:

– The choice of procedure for technique T1552: *Unsecured Credentials* has consequences on the techniques available for T1021: *Remote Services* between $(Bear, SuperUser)$ and $(Skunk, Diana)$. Indeed if T1552 rewards a SSH key to the attacker, this instance of T1021 cannot pick a procedure requiring a password (for instance a RDP connection).

**Algorithm 1** Backtracking refinement algorithm for a scenario $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$ [7]

**Require:** $\mathbf{S}^{\text{tech}} = (\mathbf{P}, \mathbf{A}^{\text{tech}})$ a scenario described at the technical level ($\mathbf{A}^{\text{tech}}$ are attack techniques)
**Require:** $\mathbf{C}$ a set of architectural constraints.
**Require:** $\{\pi_{\tau_i}\}_i$ a list of procedures instantiating each $\tau_i$
$\mathbf{S}^{\text{proc}} := (\mathbf{P}, \emptyset)$
$L_t := \mathbf{A}^{\text{tech}}$ the list of all transitions to refine, sorted to prioritize transitions requiring or rewarding secrets.
**Ensure:** $\mathbf{S}^{\text{proc}} = (\mathbf{P}, \mathbf{A}^{\text{proc}})$ a scenario described at the procedural level ($\mathbf{A}^{\text{proc}}$ are attack procedures) and $\mathbf{S}^{\text{proc}} \triangleright \mathbf{S}^{\text{tech}}$
**function** BACKTRACK($\mathbf{S}^{\text{tech}}, \mathbf{S}^{\text{proc}}, L_t, \{\pi_{\tau_i}\}_i, \mathbf{C}$)
**if** $L_t = \emptyset$ **then return** True, $\mathbf{C}$
**else**
Sort $L_t$, putting transitions requiring or rewarding secrets first.
$t = L_t[0] = ((u1, m1),(u2, m2),(\tau, \textbf{pre}, \textbf{post}))$
**if** $\{\pi_\tau\} = \emptyset$ **then return** False, $\emptyset$
**end if**
Get the set of all compatible procedures $\{Comp(t)\}_i$
**for all** $\{\pi \in Comp(t)\}$ **do**
$\mathbf{C}_{new} = \mathbf{C}$
$\mathbf{C}_{new}[m1] = \mathbf{C}_{new}[m1] \sqcup \pi[m1]$
$\mathbf{C}_{new}[m2] = \mathbf{C}_{new}[m2] \sqcup \pi[m2]$
**if** $\mathbf{C}_{new} \dashv \perp$ **then**
$\mathbf{S}^{\text{proc}}_{new} = (P, \mathbf{A}^{\text{proc}} + \pi)$
$L_{tnew} = L_t - \{t\}$
isValid, $\mathbf{C}_{final}$ = backtrack($\mathbf{S}^{\text{tech}}, \mathbf{S}^{\text{proc}}, L_{tnew}, \{\pi_{t_i}\}_i, \mathbf{C}_{new}$)
**if** isValid **then return** True, $\mathbf{C}_{final}$
**end if**
**end if**
**end for**
**return** False, $\mathbf{C}$ ▷ No valid procedure was found
**end if**
**end function**

| Technique | Possible procedures | Constraints and secrets summary for each procedure |
|---|---|---|
| $\tau_{4,8}$ = T1552 | $\pi_9$: Password in bash history | Linux, append a secret to the bash history file, rewards a plaintext password |
| | $\pi_{10}$: Private keys in .ssh | Requires Linux, creates files using a secret, rewards SSH keys |
| | $\pi_{11}$: Passwords in text file | Creates a file using a secret, rewards a plaintext password |
| $\tau_{5,9,12}$= T1021 | $\pi_{12}$: Weak SSH password | Linux, ssh service on port 22, sshd_config allowing password authentication |
| | $\pi_{13}$: SSH access from key | Linux, a ssh service on port 22, authorized_keys to be edited using a secret |
| | $\pi_{14}$: SSH access from password | Linux, a ssh service on port 22, ssh_config to allow password connections |
| | $\pi_{15}$: RDP from Password | Windows, a RDP service on port 3389, Windows registry + user password edits. |
| $\tau_7$ = T1068 | $\pi_{16}$: Dirty COW | Specific Linux version |
| | $\pi_{17}$: Vulnerable sudo version | Requires Linux, specific sudo package, edit sudoers |
| $\tau_{14}$ = T1078 | $\pi_{20}$: Password from secret | Requires the user password to match a secret |
| | $\pi_{21}$: Weak Password | Requires the user password to be easy to brute-force. |

Fig. 3: Excerpt of available procedures for each technique in the "Smash and Grab attack" scenario.

- The procedure chosen for technique T1021: *Remote Services* between *(Bear, SuperUser)* and *(Racoon, Bob)* can only be a procedure fit to require a secret, and thus cannot be "Weak SSH Password".
- Since T1078: *Valid Accounts* procedures only accept passwords, T1552: *Unsecured Credentials* between *(Raccoon, SuperUser)* and itself may not give a SSH key as a reward.
- The output architectures have varying numbers of Windows and Linux on them. Machine *Raccoon* must be on a Linux (because of our available procedures for T1068), but *Skunk*, *Badger* and *Bear* can be either.
- The number of possible resulting architectures exponentially grows as we increase the number of machines deployed, procedures implemented or amount of OS we are able to deploy. This does not however lead to computation complexity issues (running this experiment took a few seconds at most), as resolving the refinement backtracking algorithm only picks out one of these results.

## 5 Cyber-range experiment using URSID

In the previous section we discussed the procedural refinement process, which lets us refine a single scenario described on a technical level into several possible scenarios described on a procedural level. This refined description states architecture constraints, such as the required operating systems, software, files that need to be created or edited and credential values. This description (in json format) contains all the information needed to deploy the corresponding architecture, using Vagrant [7] and Ansible [11] to instantiate and populate the virtual machines, and VirtualBox [4] to deploy them. Note that generated file contents may be used to append or edit a file naturally present on the virtual machine (for instance */etc/sudoers*), or be copied as a whole. The deployment part of URSID takes such a configuration file as an input, and converts it into a Vagrantfile and several Ansible playbooks (one for each machine). This Vagrantfile is responsible for initializing VM images and network status, and each playbook describes which operations (installing software, copying files, launching services...) have to be ran on each VM. An additional layer of work is thus needed to convert this description into formats acceptable by these tools. For instance, operating system names need to be translated into virtual machine box names and services need to be installed and launched. Installing old version of packages (such as `sudo` or `python3`) may also require a dedicated script to do so, as these versions are not available on main package repositories anymore. While adding new procedures may unfortunately require additional engineering cost (especially if legacy packages are involved, or more OS versions are made available), adding procedures only requiring simple operations - such as installing a package available through traditional repositories, editing files or tweaking the version of a package- is fairly straightforward.

As a proof of concept for the uses of URSID as an educational tool, a real life experiment (denoted as CERBERE in the following) was ran at a local university by a group of 13 voluntary Master students (familiar with cybersecurity),

aiming to develop cyber attack-defense skills. This paper will focus on URSID's technical involvment in the red-team aspect of this project: a separate publication [3] details CERBERE as a whole. The scenario deployed for this occasion is shown on a technical level in Figure 4. It consists of 3 machines sharing a local network, 6 attack positions and 7 transitions showcasing 4 different techniques. The scenario requires the students to complete 7 transitions (i.e. apply 7 attack procedures) in order to access the winning attack position. The attack procedure were distributed over 3 levels of difficulty. In total, 20 variations of this scenario were generated, even though only 13 were really used. The instances differed by which procedures were picked during the refinement process, which in turn influenced the files, OS, software and account installed on the machines. For instance, the privilege escalation on machine Zagreus achieved either exploiting a vulnerability of the `sudo` package [8] or of the pkexec process [16]. All passwords and keys were also unique to each instance, making it impossible to copy other answers directly. In total, 14 procedures were implemented and out of these 10 made it into one of the instances (the others being incompatible with this specific scenario). A list of all the implemented procedures is available in the repository [2]. Each machine was also setup with logging software (in this case auditd). This was done by adding default constraints - one software to install the package and one file to tweak the configuration - for each machine during the refinement process.

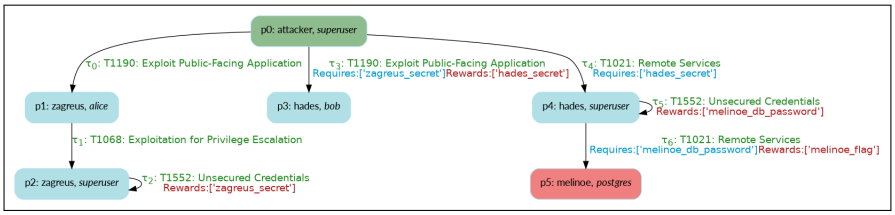


Fig. 4: Scenario "CERBERE educational CTF" at the technical level

All generated virtual machines were hosted on a single host, sporting 80 GB of RAM on 36 cores, with each machine in a given instance existing in their own virtual network. In order to give each attacker their own separate instance accessible remotely (and avoid having outside attacks/bots pollute the experiment), a double proxy was set-up. Students could only access the machine through a specific unique URL which was given to them at the beginning of the experiment.

The CERBERE scenario is simpler than the Smash and Grab scenario, as it is an exercise for students. The time allowed for this exercise was 1 hour 30 minutes. In the end, this was very little time for a single student who couldn't benefit from the help of others who had another scenario, to complete the whole exercise. For students at this level, we felt it was counter-productive to present

a scenario that was too long and contained too many unnecessary machines/accounts. Nonetheless, 4 of the 13 students managed to finish the whole scenario with hints, and 1 without hints. Additionally, 7 managed to beat the first out of all 3 machines. This analysis was done after the experiment by recovering and analyzing the auditd log files for every machine. Since we also had access to the procedural-level description of each scenario, the log analysis process was partly automatized by writing customized SIGMA [10] rules for each procedure. This let us know for each instance which transitions were successfully attacked and when.

The publicly accessible version of the CERBERE instances has since been shut down. However, the URSID repository contains instructions to deploy an instance locally, alongside directions on how to attack it. Note that the networking setup in this local deployment will differ from the one that was used during the experiment, as machines will be deployed on a local virtual network as opposed to a public one.

## 6 Related works

Cyber-range and cyber-training generation tools have varying scopes in the literature, ranging from being used as a learning tool for academic students, to being a training platform for AI-based attackers. In particular, recent researches did tackle the issue of cyber-range re-usability and deployment based on a scenario description. VulnerVAN [21] is a tool aiming to generate vulnerable architectures ready to be deployed for red team training, based on a user provided description. The user describes the network architecture and an attack sequence, which is represented as a sequence of MITRE ATT&CK techniques [13]. The work of Yamin et al [22] and Costa et al [5] also converts scenarios into cyber-range architectures using formal description languages (such as Datalog) and both also incorporate a notion of constraints between exploits which needs to be checked before validating a scenario. While not advertising itself as a cyber-range per se, SecGen [19] is a publicly available vulnerable virtual machine deployment tool made for educational purposes, with the goal of being a Capture-The-Flag and student lab deployment platform. It relies on low-level XML description of architectures (and which specific exploits to implement) to be deployed. While all of these tools are of immense use to researchers, it is to be noted that they tend to rely on low-level description of attack scenarios, all requiring specific information about which exploits to implement and software to install before being able to deploy a scenario. VulnerVAN and the work of Yamin et al [22] come close, but do not go as far as deploying several scenarios corresponding to a single high-level description. We believe URSID's formalization main perk to be its genericity, and URSID's deployment main interest to be the refinement process. The ability to deploy architectures with only a high-level description (once the preliminary work of writing out procedure constraints is achieved) seems fairly unique in the literature so far, and an interesting field of research to expand in the future.

## 7 Conclusion and future works

In this work we presented URSID, an automatic cyber-range generating tool with a new approach toward attack scenario and insecure architecture deployment. Starting from a formal representation at the technical level, URSID refines a single scenario into several low level description (at the procedural level), which represent the same overall attacker scenario but all differ procedure wise. All these low level descriptions may then be deployed in virtual machine networks and attacked following the same path as described in the high level description, but with varying exploits. While there is an unavoidable engineering cost in the implementation of procedures and the translation of low level descriptions into virtual machine architectures, given a rich enough database of procedures, enhancing, tweaking, or re-deploying a given scenario has a very low cost. URSID was tested as part of a public training Capture-The-Flag experiment in a university, where its refinement abilities meant students were all following the same overall attack path while all having a unique experience. As future works, we first aim to automatize the process of populating a network with additional machines even more, by making it possible to add prefab machines instead of having to write every single attack position manually. Scenarios would thus be a combination of a written-by-hand attack path and automatically generated machine clusters to populate the network. We are also working on combining URSID with pattern matching for system logging tools, such as SIGMA [10] rules. Indeed, associating pattern matching with procedures could let the user automatically track in real time the path of an attacker through the scenario, by checking for known exploit patterns which were specifically implemented in this instance. Finally more practical experiments involving URSID are on their way, first an improvment upon the CERBERE experiment, then tests of URSID's abilities as a honeynet generation platform.

## References

1. Ammann, P., Wijesekera, D., Kaushik, S.: Scalable, graph-based network vulnerability analysis. In: Proceedings of the 9th ACM Conference on Computer and Communications Security. p. 217–224. CCS '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/586110.586140, https://doi.org/10.1145/586110.586140
2. BESSON, P.V.: Ursid repository (2023), https://gitlab.inria.fr/pibesson/ursid-final
3. Besson, P.V., Brisse, R., Orsini, H., Talon, N., Lalande, J.F., Majorczyk, F., Sanchez, A., Viet Triem Tong, V.: CERBERE: Cybersecurity Exercise for Red and Blue team Entertainment, REproducibility. In: CyberHunt 2023 - 6th Annual Workshop on Cyber Threat Intelligence and Hunting. IEEE Computer Society, Sorrento, Italy (Dec 2023), https://centralesupelec.hal.science/hal-04285565
4. Corporation, O.: Virtualbox (2005), https://www.virtualbox.org/
5. Costa, G., Russo, E., Armando, A.: Automating the generation of cyber range virtual scenarios with VSDL. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications **13**(4), 61–80 (dec 2022). https://doi.

org/10.58346/jowua.2022.i4.004, https://doi.org/10.58346%2Fjowua.2022.i4.004
6. FBI: Internet crime report 2021 (2021), https://www.ic3.gov/Media/PDF/AnnualReport/2021_IC3Report.pdf
7. HashiCorp: Vagrant (2010), https://www.vagrantup.com/
8. Hat, R.: Cve-2019-14287 (2019), https://access.redhat.com/security/cve/cve-2019-14287
9. Hemberg, E., Kelly, J., Shlapentokh-Rothman, M., Reinstadler, B., Xu, K., Rutar, N., O'Reilly, U.M.: Linking threat tactics, techniques, and patterns with defensive weaknesses, vulnerabilities and affected platform configurations for cyber hunting (2021)
10. HQ, S.: Generic signature format for siem systems (2017), https://github.com/SigmaHQ/sigma
11. Inc, A.: Ansible (2012), https://www.ansible.com/
12. Mensah, P.: Generation and Dynamic Update of Attack Graphs in Cloud Providers Infrastructures. Ph.D. thesis, CentraleSupélec, Châtenay-Malabry, France (2019), https://tel.archives-ouvertes.fr/tel-02416305
13. MITRE: The mitre att&ck matrix for enterprise (2018), https://attack.mitre.org/matrices/enterprise//
14. MITRE: Apt29 adversary emulation (2021), https://github.com/center-for-threat-informed-defense/adversary_emulation_library/tree/master/apt29
15. NIST: Cyber ranges (2018), https://www.nist.gov/system/files/documents/2018/02/13/cyber_ranges.pdf
16. NIST: Cve-2021-4034 detail (2021), https://nvd.nist.gov/vuln/detail/cve-2021-4034
17. Outkin, A.V., Schulz, P.V., Schulz, T., Tarman, T.D., Pinar, A.: Defender policy evaluation and resource allocation using mitre attck evaluations data (2021)
18. Russo, E., Costa, G., Armando, A.: Scenario design and validation for next generation cyber ranges. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). pp. 1–4 (2018). https://doi.org/10.1109/NCA.2018.8548324
19. Schreuders, C.: Security scenario generator (secgen), https://github.com/cliffe/secgen (2017), https://github.com/cliffe/SecGen
20. Sharma, Y., Birnbach, S., Martinovic, I.: Radar: A ttp-based extensible, explainable, and effective system for network traffic analysis and malware detection (2023)
21. Venkatesan, S., Youzwak, J.A., Sugrim, S., Chiang, C.Y.J., Poylisher, A., Witkowski, M., Walther, G., Wolberg, M., Chadha, R., Newcomb, E.A., Hoffman, B., Buchler, N.: Vulnervan: A vulnerable network generation tool. In: MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM). pp. 1–6 (2019). https://doi.org/10.1109/MILCOM47813.2019.9021013
22. Yamin, M.M., Katt, B.: Modeling and executing cyber security exercise scenarios in cyber ranges. Computers & Security **116**, 102635 (2022). https://doi.org/https://doi.org/10.1016/j.cose.2022.102635, https://www.sciencedirect.com/science/article/pii/S0167404822000347