POTASSIUM: Penetration Testing as a Service

Richard Li, Dallin Abendroth, Xing Lin, Yuankai Guo
Hyun-wook Baek, Eric Eide, Robert Ricci, Jacobus Van der Merwe

University of Utah
Salt Lake City, UT, USA

licai@cs.utah.edu Dallin.Abendroth@utah.edu
{xinglin, guoyk, baekhw, eeide, ricci, kobus}@cs.utah.edu

Abstract

Penetration testing—the process of probing a deployed system for security vulnerabilities—involves a fundamental tension. If one tests a production system, there is a real danger of collateral damage; this is particularly true for systems hosted in the cloud due to the presence of other tenants. If one tests against a separate system brought up to model the live one, the dynamic state of the production system is not captured, and the value of the test is reduced. This paper presents POTASSIUM, which provides *penetration testing as a service* (PTaaS) and resolves this tension for system owners, penetration testers, and cloud providers. POTASSIUM uses techniques originally developed for live migration of virtual machines to clone them instead, capturing their full disk, memory, and network state. POTASSIUM isolates the cloned system from the rest of the cloud, providing confidence that side effects of the penetration test will not harm other tenants. The penetration tester effectively owns the cloned system, allowing testing to be more thorough, efficient, and automatable. Experiments with our POTASSIUM prototype show that PTaaS can detect real-world vulnerabilities while having minimal impact on cloud-based production systems.

Categories and Subject Descriptors K.6.5 [*Management of Computing and Information Systems*]: Security and Protection—unauthorized access; D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools

General Terms Experimentation, Security

Keywords cloud computing; OpenStack; pentesting; PTaaS

The definitive version was published in *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, Kohala Coast, HI, USA, Aug. 2015, `http://dx.doi.org/10.1145/2806777.2806935`

1. Introduction

Security vulnerabilities are a fact of life in networked systems. Their causes range from application bugs to operating system loopholes to misconfigurations—even of the appliances designed to protect the network [7, 18, 39]—and more. Given this reality, penetration testing *(pentesting)* has become a critical tool for protecting networked systems. The goal of a pentest is to uncover vulnerabilities in a target system so that the owner of that system can take appropriate action to correct them. A pentester acts like a real attacker, attempting to compromise the system ("penetrate" it) in order to learn the effectiveness of particular attacks. The pentester is likely to employ standard "hacking" tools to check for well-known vulnerabilities, and sophisticated tests may include attacks that are custom-made for the specific target system.

Pentesting comes with a fundamental tension. On one hand, running a pentest on a live, production system is ideal, because doing so captures the exact dynamic state of the system. On the other hand, doing so carries a real risk of damage, causing the system to be unavailable to users (if, say, a DoS attack succeeds) or losing data (e.g., if the system crashes). Penetration testers today are thus left with two choices, neither of which is ideal: test the production system and run these risks, or set up a model of the system that may not have exactly the same vulnerabilities. For example, if an administrator has inadvertently made a configuration change to the production system that weakens it, but has failed to document this change, it would be easy to miss the change when setting up the model. Similarly, user behavior can affect the state of the system in important ways. For example, users with poor passwords may provide an initial vector for attack, or the system may have been previously compromised by a real attacker who left a backdoor.

Things become even more problematic in cloud environments. The multi-tenant nature of a cloud means that tests can affect not only the production system, but potentially other, unrelated tenants as well. For example, a DoS attack could exhaust network resources shared by many tenants, or

attacks on the hypervisor may compromise other VMs. To protect their infrastructure, public cloud providers typically require that pentests be registered with them in advance, and conform to certain types of attacks (for example, not attempting DDoS) [2]. Additionally, the time period allowed for pentesting is limited. For example, AWS has a three-month time limit for each pentesting authorization [2]. While these precautions are reasonable for the cloud provider to take, the result is that pentesting in the cloud is a cumbersome and expensive operation, and not something that can be done frequently.

To resolve these tensions, we propose *penetration testing as a service* (PTaaS). With PTaaS, the cloud provider plays a role in enabling the pentest, offering benefits for all three participants: the system owner, the pentester, and the cloud provider. The cloud provider (likely for a fee) leverages their control over the hypervisor, network, and storage system to create an exact clone of the production system to be tested. The clone gives the pentester a copy of the running system with the same dynamic state, meaning that it will exhibit the same vulnerabilities as the production system. The owner of the production system does not need to worry about the pentest disrupting normal operations of the system, and the cloud provider can place the clone in a part of its infrastructure that will not disrupt other tenants.

We have developed a prototype PTaaS system called POTASSIUM, which is implemented as an extension to OpenStack [30]. POTASSIUM takes techniques originally developed for live migration of virtual machines and repurposes them for cloning: it "migrates" VMs, but keeps the originals running instead of terminating them. The original system continues to run and serve clients, who see only a modest performance slowdown for the few seconds that cloning takes to complete. The cloned copy is placed in an isolated environment, to which only the pentester has access. Because many cloud-based systems are comprised of many VMs, POTASSIUM creates consistent snapshots of groups of VMs. POTASSIUM includes a proof-of-concept, fully automated pentester based on the Metasploit Framework [32], but we envision that POTASSIUM opens up a marketplace for a range of pentesting services in the cloud, from simple, automated tests to customized, in-depth campaigns performed by human experts.

We make the following contributions. First, we introduce the notion of PTaaS and identify the value of having the cloud provider participate in pentesting: namely, that it simultaneously preserves the *availability* and *integrity* of the production system, the *validity* of the tests run by the pentester, and the *safety* of the cloud. Second, we design an architecture for a PTaaS system, POTASSIUM, that clones entire systems and places the clones in isolated environments suitable for pentesting. Third, we present a prototype of POTASSIUM and use it to demonstrate a proof-of-concept, automated pentester. We show that POTASSIUM scales to systems comprising dozens of VMs and imposes little impact on the production system.

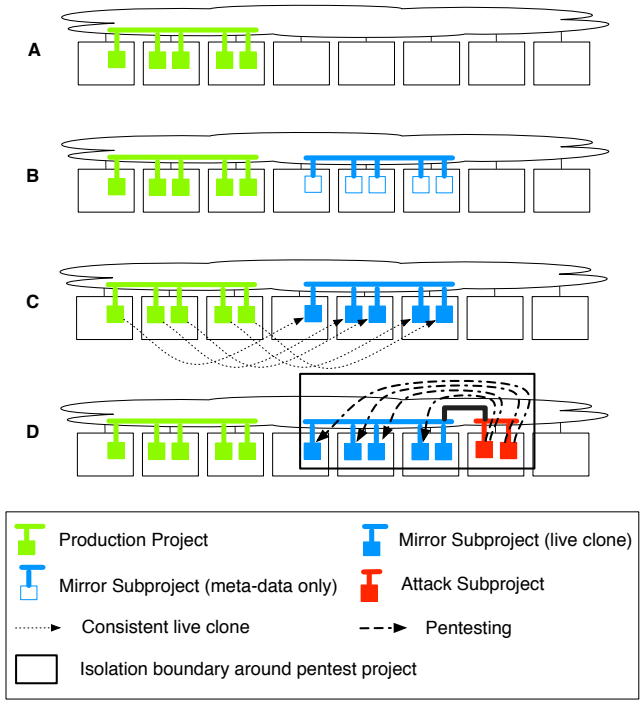

Figure 1. POTASSIUM's workflow. POTASSIUM clones a production project in the cloud for the purpose of pentesting.

2. POTASSIUM Architecture

To set up and perform a penetration test, POTASSIUM performs the steps illustrated by the example depicted in Figure 1. The initial state of the cloud is represented in step A: a tenant has allocated a collection of resources including several VMs and a network. Following OpenStack convention, we refer to such a collection of resources as a *project*; we refer to a user-deployed project as a *production project*. In step B, POTASSIUM creates a new project: using the standard APIs of the cloud management system, it obtains a description of the production project and creates a new project matching that description. We refer to the copy as the *pentest project*. At this point, the pentest project has the same structure as the production project, but not the same internal state. In step C, POTASSIUM creates a consistent snapshot of the production project—including all VM memory contents, disk contents, and network packets in flight—and inserts that state into the pentest project. Finally, in step D, POTASSIUM allocates "attacking" resources, adds them to the pentest project, and performs the pentest. The pentest project now contains two parts: the *mirror subproject*, which is the set of resources that mirror the production project, and the *attack subproject*, which is the set of resources introduced for pentesting. ("Subproject" is not a standard OpenStack term; we introduce the terminology to distinguish between the mirrored resources and the attacking resources.) POTASSIUM ensures that the pentest project is isolated so that the effects of the penetration test cannot escape. Step D in Figure 1 illustrates an attack subproject that emulates an external attacker.

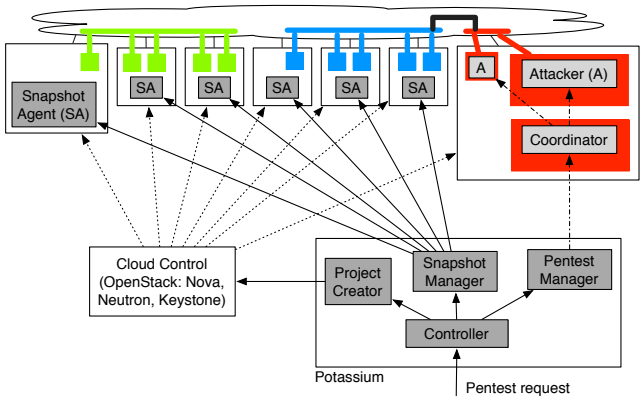

Figure 2. POTASSIUM's architecture. POTASSIUM's servers communicate with cloud management services as well as agents and attackers running on the cloud's compute nodes.

POTASSIUM can insert attacking VMs into the mirror subproject's internal networks, and "commandeer" VMs within the mirror subproject, to emulate internal attacks.

The software architecture that realizes the workflow is shown in Figure 2. There is one instance of POTASSIUM that manages all pentesting requests for projects within the cloud; in this sense, POTASSIUM is an extension of the cloud infrastructure. It contains a component responsible for each of the workflow stages described above as well as an overall *Controller* that receives pentesting requests and orchestrates the workflow. The *Project Creator* is responsible for (standard) cloud-related actions, i.e., creating pentest projects, creating mirror subprojects from production projects' metadata, creating attack subprojects, interconnecting mirror and attack subprojects, and properly isolating pentest projects as a whole. The Project Creator interacts with the standard cloud control architecture: in an OpenStack cloud, this involves interacting with Nova, Neutron, Keystone, and other OpenStack components. The *Snapshot Manager* and the *Snapshot Agents* are responsible for replicating the state of a production project within a mirror subproject. Finally, the *Pentest Manager* orchestrates the actual pentesting process. Specifically, every attack subproject contains a *Coordinator* that manages the *Attackers* within the subproject. The Pentest Manager is POTASSIUM's interface to the Coordinators.

POTASSIUM's software architecture is motivated by several design principles that are necessary to enable penetration testing as a service (PTaaS). These include *validity*, *safety*, *availability*, *scalability*, and *extensibility*.

Validity. First and foremost, the penetration-testing results obtained by PTaaS must be valid: they should provide the same information as a pentest performed against the original production project. To address this principle, POTASSIUM's Project Creator, Snapshot Manager, and Snapshot Agents cooperate to replicate the full state of the production project within (the mirror subproject of) a pentest project.

POTASSIUM creates the mirror in two steps, as previously illustrated in Figure 1. The first step is straightforward: the Project Creator obtains the metadata of the production project and invokes the cloud platform to create a mirror with the same metadata. The details of this process are described in Section 3.1. The second step, recreating the full state of the VMs and network in the production project, is more challenging. To implement this step, the Snapshot Agents perform *live, consistent checkpointing* over the production project. Consistency means that the state of the production project is captured at a single, logical instant of time: in particular, once a VM has completed its snapshot, any packet that it sends will not be delivered until the recipient VM has also completed its snapshot. Our implementation of consistent, live checkpointing is presented in Section 3.2.

Safety. PTaaS must be safe in the sense that pentesting activity must not affect production projects or other systems on the Internet. This includes all the activity within the pentest project: not only the attacks from the attack subproject, but also the activity of the mirror subproject, normal or otherwise. This challenge is addressed by the Project Creator, which uses the standard APIs of the cloud platform to disconnect the pentest project from any other network—save for an access route that allows POTASSIUM's Pentest Manager to communicate with the attack Coordinator within the pentest project (Section 3.4). Only the Coordinator is allowed to send traffic outside of the pentest project, and it is configured not to relay traffic between the "inside" and the "outside" of the pentest project.

POTASSIUM trusts the cloud platform to implement its network-configuration features correctly. POTASSIUM also trusts the cloud-provided hypervisors to operate correctly. POTASSIUM is not intended for pentests that seek to compromise the underlying cloud platform or hypervisors.

Availability. PTaaS seeks to have a low impact on the availability and performance of the production project. To meet this goal, POTASSIUM does two things. First, the Project Creator can place pentest projects on physical resources that are separate from those used by production projects (Figure 1). This is possible using standard cloud APIs such as availability zones (Section 3.4). Second, the Snapshot Agents perform live consistent checkpointing, as previously mentioned. Live checkpointing [10] allows the production system to execute while it is being checkpointed: while performance may be reduced during the short time it takes to checkpoint, the project remains available to clients.

Scalability. POTASSIUM's architecture allows pentesters to take advantage of the scalability provided by clouds in two ways. First, POTASSIUM can manage multiple pentest projects at once. These might correspond to independent production projects—i.e., two separate users, who happen to be running pentests at the same time—or they might correspond to concurrent pentests over a single production

project. Second, POTASSIUM implements multiple strategies for allocating and positioning Attackers against a mirror subproject (Section 3.3), e.g., to emulate both external and internal attacks. The ability to allocate large numbers of attacker VMs allows POTASSIUM to trade space for time, by performing pentests against multiple hosts in parallel.

Extensibility. Our prototype POTASSIUM implementation uses the Metasploit Framework [32] to perform actual pentesting. Our architecture is, however, not specific to Metasploit. Its design is amenable to different choices for pentesting tools; the Coordinator serves as an adapter between POTASSIUM's Pentest Manager and the implementations of the Attackers.

More generally, the separation between POTASSIUM's "core" and the Coordinators and Attackers supports realizations of POTASSIUM that support various business relationships. We have described POTASSIUM so far in the context of a cloud provider that itself provides the complete PTaaS to its customers. In an alternative model, the cloud provider might simply provide the low-level mechanisms (consistent project checkpointing and isolation) and allow third parties to perform the actual penetration testing.

3. Implementation

We implemented a prototype of the POTASSIUM architecture within the OpenStack cloud environment, using the OpenStack Juno release [30]. While we were able to implement significant parts of POTASSIUM using unmodified OpenStack APIs, our implementation of live, consistent checkpointing required modifications to the QEMU hypervisor [6] and the "libvirt" virtualization API [35] that are underlying components of OpenStack. Our POTASSIUM prototype uses the Metasploit Framework [32] to perform actual pentesting.

3.1 Mirror Subproject Creation

To ensure that the pentesting performed within a pentest project produces valid results—representing potential vulnerabilities in the original, production project—POTASSIUM requires two things. First, within the pentest project, the mirror subproject's configuration must be an exact replica of the production project's configuration. Second, the mirror subproject's state must be initialized from a consistent checkpoint of the VMs in the production project. This section details the implementation of POTASSIUM's Project Creator, which addresses the first requirement. (The implementation of consistent checkpointing is detailed in Section 3.2.)

The basic steps of creating the mirror subproject are straightforward. The Project Creator invokes OpenStack's standard APIs to obtain all the relevant metadata associated with the production project, and it then invokes those APIs again to create the pentest project and its mirror subproject, which has the same configuration as the production project. The metadata handled by the Project Creator includes user accounts, network quotas, compute quotas, network and subnet IDs, router configuration, port information, security groups, and all the information related to VMs. The Project Creator obtains this metadata from three standard OpenStack services: identity services (Keystone), networking services (Neutron), and compute services (Nova).

OpenStack uses its own internal conventions in assigning (unique) IDs to cloud components. To properly copy the configuration of the production project, the Project Creator maintains a mapping between the IDs used in the production project and the corresponding component IDs used in the mirror subproject. For example, a subnet in the production project will be associated with a specific network. By consulting its mapping from production component IDs to mirror component IDs, the Project Creator can ensure that this relationship is maintained in the mirror.

Every VM in the mirror subproject is a clone of a VM in the production project. For a VM in the mirror subproject to function properly, it must be created with the same IP and MAC layer addresses that its original had in the production project. (Note that, in the production project, the assigned IP and MAC addresses are typically unimportant and might simply be allocated via automated mechanisms such as DHCP.) The Project Creator explicitly creates ports with the appropriate attributes to ensure this consistency.

To restore a suspended VM (i.e., continue its execution), OpenStack finds the VM's state by reading a file. In addition to the VM's memory contents, the file also contains metadata that identifies the VM instance, its virtual NICs, and so on. The files created from snapshotting the production project include the IDs of components in the production project; to use these files for VMs in the pentest project, POTASSIUM must replace the identifiers within the files to refer to components of the pentest project.

The Project Creator achieves this in two steps as follows. First, the Project Creator builds the mirror subproject by copying the configuration of the production project. Note that the mirror's VMs initially refer to the "base images" that were used to start the VMs within the production project—*not* the current snapshots of those VMs. The Project Creator suspends the VMs in the mirror subproject, and it extracts the relevant component IDs from the metadata of those suspended VMs. Second, to "retarget" the snapshot images of the production-project VMs into the corresponding mirror-subproject VMs, the Project Creator copies the snapshot image files. In the copied files, it replaces the component IDs—which refer to components of the production project—with the IDs that refer to the corresponding components within the mirror subproject. To perform snapshotting and retargeting quickly, the Project Creator uses multithreading to perform work in parallel: taking snapshots, copying image files, and replacing IDs. At this point, the VMs in the mirror subproject can be restored via standard OpenStack mechanisms, and they execute the cloned state from the production project.

OpenStack supports "floating IP addresses," which are public IP addresses that can be dynamically assigned to VMs.

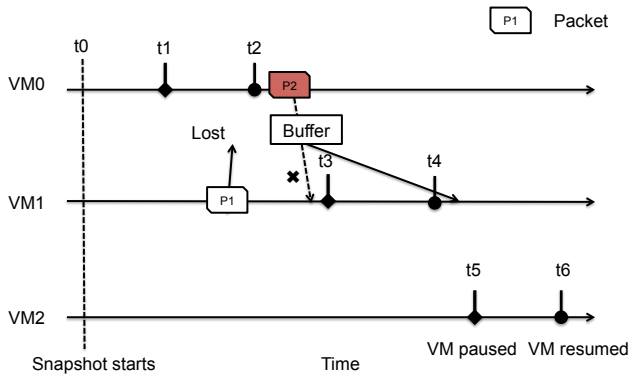

Figure 3. Example checkpoint timeline. For live, consistent checkpointing, packet *P2* must be delivered after time *t4*.

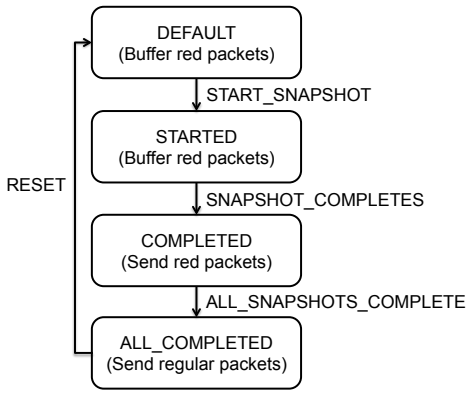

Figure 4. VM state machine for consistent checkpointing.

When the Project Creator creates the pentest project, it does *not* copy any of the production project's floating IP addresses, because doing so would lead to obvious conflicts. Moreover, because the pentest project is intended to be isolated from the Internet, there is no purpose to copying the production project's floating IPs. Practically speaking, this means that the production project's active sessions with external hosts will not be maintained in the mirror subproject (a desired outcome) and they will continue in the production project (also a desired outcome).

POTASSIUM creates closed, isolated pentest projects (Section 3.4) in order to satisfy its design principles. In our current implementation, POTASSIUM clones only the resources that are *internal* to a production project: e.g., VMs and their local storage. To bring *external* resources (such as cloud-wide storage or database services) into this closed system, it would be necessary to create exact, isolated copies of them that would be accessible to pentest projects. For some services, such as block storage, this may be straightforward to achieve using existing snapshot mechanisms, while for others, it may be infeasible. We leave the exploration of these topics as future work.

3.2 Live Consistent Checkpointing

POTASSIUM's Snapshot Manager and Snapshot Agents implement a live, consistent checkpointing algorithm that creates *snapshots* of a production project. *Live* means that the snapshot is taken transparently; *consistent* means that the final snapshot captures the state of the production project at a single, logical instant of time. If POTASSIUM were to capture the states of individual VMs in isolation, without regard to their ongoing communication, the resulting snapshot could be inconsistent. We explain this problem with the aid of Figure 3, which also depicts the solution that POTASSIUM implements to prevent this.

The figure shows an example timeline in which POTASSIUM creates a snapshot for a project containing three VMs. At time *t0*, the checkpointing algorithm begins: POTASSIUM's Snapshot Manager issues commands to the Snapshot Agents, which in turn command the VMs to snapshot themselves. Each VM starts to save its memory state by performing iterative memory copying, a technique previously developed for live VM migration [10]. The time required to complete the snapshot may vary across the VMs, depending on their memory utilizations and CPU loads. In this example, VM0 pauses itself at time *t1* and starts to save its final dirty memory pages and create a snapshot of its disk. (Packets such as *P1*, sent to a paused VM, are dropped; we revisit this issue later.) When VM0 completes its snapshot at time *t2*, it resumes normal execution. VM1 and VM2 perform the same steps, but at different times.

Without a consistent snapshot mechanism, when VM0 resumes from its snapshot at time *t2*, it may start to send packets such as *P2* to VM1. As shown by the dashed arrow in the figure, VM1 receives this packet while it is still creating its snapshot. Thus, in the state represented by VM1's snapshot (captured at *t3*), VM1 has already received packet *P2*. In VM0's snapshot, however, *the packet has not yet been sent*. The snapshots of the two VMs are thus inconsistent.

To create consistent snapshots over a collection of VMs, any packets sent by VMs that have completed their snapshots must not be delivered to VMs that have *not* completed their snapshots. We refer to such packets as *inconsistent packets* hereafter. An inconsistent packet must either be (1) dropped or (2) buffered and released to the receiving VM only after the receiving VM completes its snapshot.

For our POTASSIUM prototype, we implemented a consistent, distributed snapshot protocol within QEMU and libvirt, based on approaches proposed in VNSnap [21] and hotSnap [12]. Our implementation uses QEMU's live-snapshot mechanism to independently take a live snapshot for each individual VM in the production project. It uses packet coloring and buffering to deal with inconsistent packets.

In our distributed live-snapshot implementation, each VM in a production project is associated with an instance of the state machine shown in Figure 4. A VM begins in the *DEFAULT* state. When POTASSIUM needs to checkpoint a production project, the Snapshot Manager sends a *START_SNAPSHOT* command to each VM, via the Snapshot Agents. Each VM transitions to the *STARTED* state and be-

gins to take its snapshot. When a VM completes its snapshot, it transitions to the *COMPLETED* state. The Snapshot Manager periodically checks the status of each VM, and when all have completed their snapshots, the Snapshot Manager sends an *ALL_SNAPSHOTS_COMPLETE* command to every VM. Each VM then transitions to the *ALL_COMPLETED* state.

Our implementation uses message coloring and buffering to achieve consistent distributed snapshots [12]. Any packet sent from a post-snapshot VM (i.e., one that has completed its snapshot) is marked as a "red packet" by setting a bit in the EtherType field of the Ethernet packet header. (That bit is not normally set for IP-over-Ethernet packets.) When a VM finishes its snapshot and is in the *COMPLETED* state, it starts to send red packets (e.g., *P2* in Figure 3). When a VM receives a red packet but has not yet completed its snapshot (in the *DEFAULT* or *STARTED* state), the red packet is buffered. The red packet is released to the receiving VM only when it has finished its snapshot.[1] When all VMs finish their snapshots and transition to the *ALL_COMPLETED* state, they resume sending regular packets. POTASSIUM's Snapshot Manager resets the VMs to the *DEFAULT* state when there are no more red packets in the network.

In our POTASSIUM prototype, packets that are sent to a paused VM during checkpointing—e.g., *P1* in Figure 3—are dropped. We chose not to handle these packets in a special way in our prototype because (1) pause times are usually quite short and (2) applications generally either use reliable transmission protocols such as TCP or are tolerant with respect to packet loss. While lost packets may have an impact on the production project, they do not affect the consistency of the snapshots created by our POTASSIUM prototype.

3.3 Pentesting Modes

By exploiting the flexibility of the cloud platform and the fact that pentesting is always performed on a (throwaway) copy of the production project, POTASSIUM can realize a variety of pentesting modes or scenarios. The modes implemented in our POTASSIUM prototype are depicted in Figure 5. In all modes, POTASSIUM injects an *attack subproject* into the project that is to be pentested. The attack subproject encompasses a group of Attacker VMs, connected to a Coordinator via a control network, that are the "launch points" for the pentest.

The top diagram in Figure 5 depicts *internal mode*. POTASSIUM creates multiple Attackers and attaches one to each network within the mirror subproject. In this way, the Attackers can directly perform penetration testing on VMs in the mirror subproject, regardless of whether those VMs are reachable from an external network. This mode is useful for understanding the overall vulnerabilities in the project.

[1] To ensure that packets are released to the VM in the order they were received, POTASSIUM buffers *all* packets to the VM once the VM has received a red packet.

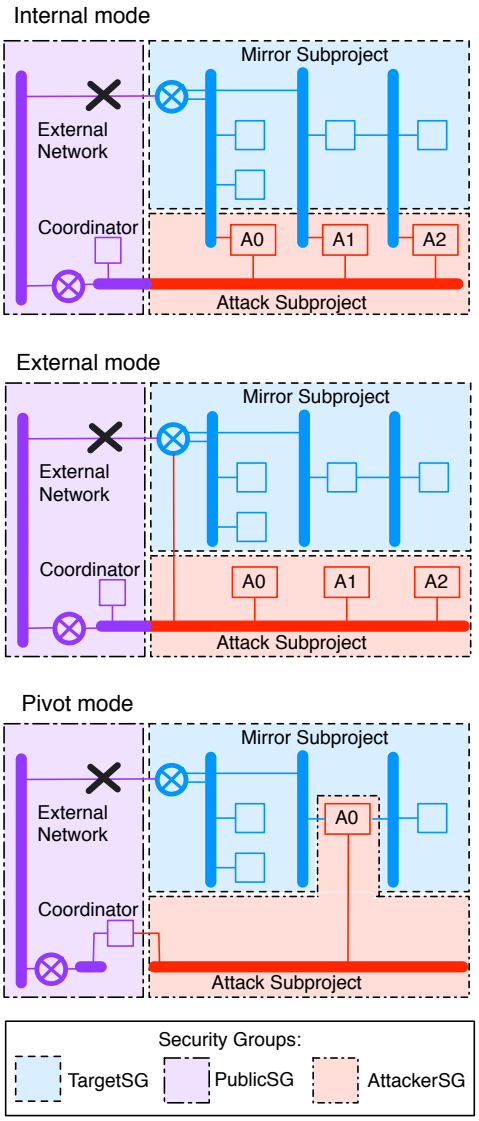

Figure 5. Pentesting modes. POTASSIUM uses security groups to control traffic within a pentest project. In addition, the mirror subproject is cut off from the external network.

The middle diagram in Figure 5 shows *external mode*. In this case, the attack subproject is attached to the mirror subproject so as to emulate an external attacker. This mode is therefore useful for understanding how well the project's defenses stand up against an external attacker. External mode might be used, for example, to test the correctness of security group rules that are deployed to defend against a known vulnerability in the project that cannot otherwise be fixed.

The bottom diagram in Figure 5 depicts *pivot mode*. Here, pentesting is performed in multiple rounds from Attackers that replace VMs in the mirror subproject. This mode imitates the way an intruder is able to attack new targets from the point of view of an already compromised VM [8]. In addition to potentially discovering actual chained vulnerabilities in the project, this mode is also useful to perform "what-if" analyses:

i.e., understanding which parts of a multi-tier setup might be vulnerable if the front tier is compromised.

3.4 Isolated Pentesting

Our POTASSIUM implementation employs multiple isolation mechanisms to ensure that pentesting activities do not impact other cloud tenants. First, when creating a pentest project, POTASSIUM selects an availability zone separate from the ones where production projects are created. This ensures that pentest projects are not co-resident with production projects.

Second, as shown in the diagrams of Figure 5, connectivity between the mirror subproject and the external network is disabled. This prevents traffic from the mirror subproject—normal, or related to pentesting—from "leaking" out to the external world via its normal path to the Internet.

Third, our POTASSIUM implementation uses OpenStack security groups to allow the Attackers to be controlled by the Coordinator, and to allow Attackers to reach VMs in the mirror subproject, while preventing Attackers from reaching the external network. The dash-outlined regions in Figure 5 depict the application of the different security groups in each of the three pentesting modes.

- The Coordinator is created with a *PublicSG* security group, allowing the Coordinator to send and receive traffic freely. This allows the Coordinator to be reachable from POTASSIUM's Pentest Manager (Figure 2).
- The VMs within the mirror subproject are associated with a *TargetSG* security group, in addition to their normal security groups that were cloned from the production project. *TargetSG* acts as a tag that can be used to easily allow connectivity between the attack subproject and the mirror subproject, without impacting the original security group rules associated with the mirror subproject VMs.
- Attackers are associated with an *AttackerSG* security group. *AttackerSG* is only allowed to talk to *PublicSG* and *TargetSG*. This enables Attackers to receive pentesting commands from, and return results to, the Coordinator.

In pivot mode, security groups alone are insufficient to prevent Attackers from communicating with the external network. When an Attacker stands-in for a VM in the mirror subproject, it effectively "inherits" that VM's security groups, which may allow access to the Internet. To prevent this, POTASSIUM creates an additional network that links only the Coordinator and the router to the outside world. In this way, the Attacker can at most reach the Coordinator, even if its inherited security groups would allow it to go further.

3.5 Automated Pentesting

Our POTASSIUM prototype uses the Metasploit Framework (MSF) [32] to perform automatic penetration testing. MSF is the open-source version of Metasploit, the most popular pentesting software environment today. It comes with literally thousands of modules for discovering and exploiting vulnerabilities in real-world systems. New modules are continually added to MSF as new vulnerabilities are found [31].

In our POTASSIUM implementation, the pentesting process is started by a call from the Pentest Manager to the pentest project's Coordinator. The Pentest Manager sends information about the mirror subproject, including the IP addresses of the VMs, and an "assignment sheet" that directs Attackers to particular VMs within the mirror subproject. The Coordinator then relays commands to the Attackers, which run in parallel. The Coordinator collects results by asking Attackers for their "sessions" on the target VMs: a session is a state, created by an exploit, in which a real attacker could execute arbitrary code. At the end of pentesting, the Coordinator generates a report that catalogs the discovered vulnerabilities. For each successful exploit, the data in the report includes MSF's name for the exploit, its description, and the payload used to open the session. The report may also include the privileges obtained by a successful exploit and the service or application version targeted by the exploit.

Automated pentesting is a valuable, fast, and inexpensive way to find vulnerabilities—but it is also limited. A tool like MSF cannot replicate a human pentester's ability to make use of discovered information (e.g., user names and keys) and to perform new combinations of actions that lead to compromises. We envision, but have not yet implemented, additions to POTASSIUM to aid human-guided pentesting. In these additions, systems like MSF would be tools in the human expert's arsenal. We have consulted with professional pentesters at our institution, and they told us that POTASSIUM's ability to create isolated copies of production environments for testing would be a "lifesaver" for their work [17]. More generally, as a cloud-provided service, POTASSIUM potentially creates a market for a range of pentesting services, from simple, automated tests to in-depth, human-driven analyses.

4. Evaluation

We evaluated our POTASSIUM prototype using an OpenStack Juno instance deployed within the Utah Emulab network testbed [36]. We measured the end-to-end time to perform POTASSIUM-style pentesting as a function of the number of VMs in the production project (Section 4.1). We evaluated the performance impact of the checkpointing procedure on a production system in terms of web-server response delay (Section 4.2). We validated the effectiveness of our live, consistent checkpointing implementation by comparing it to a per-VM (inconsistent) checkpointing procedure (Section 4.3). Finally, we tested POTASSIUM's ability to reveal vulnerabilities with two case studies (Section 4.4 and Section 4.5). Our overall conclusions are that our prototype scales well for production systems comprising up to 100 VMs; that the impact of live, consistent checkpointing on the production system is limited and similar to the impact of live, non-consistent checkpointing; and that our prototype can successfully detect security vulnerabilities in our test cases.

We set up our OpenStack environment on seven physical Emulab nodes configured in a LAN. One served as the

Production System Size	**1**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
Number of Attackers	**1**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**
Mirror Creation	35.20	81.83	136.43	189.90	234.92	270.05	337.69	382.18	445.79	511.89	556.38
Metadata Cloning	*2.70*	*10.05*	*18.64*	*26.78*	*36.00*	*45.42*	*53.43*	*62.33*	*72.23*	*80.11*	*87.38*
Taking Snapshot	*2.56*	*3.41*	*4.87*	*5.48*	*7.07*	*8.26*	*9.68*	*10.05*	*12.51*	*12.39*	*13.83*
Copying files	*16.74*	*37.82*	*59.38*	*81.44*	*105.10*	*120.81*	*167.24*	*178.84*	*226.12*	*271.23*	*298.78*
Replacing IDs	*13.20*	*30.55*	*53.54*	*76.20*	*86.75*	*95.56*	*107.34*	*130.94*	*134.94*	*148.16*	*156.39*
Attacker Creation	14.31	25.73	44.04	65.84	92.34	116.94	151.64	190.31	234.11	282.79	334.42
Penetration Testing	11.00	41.52	48.00	49.00	48.65	49.00	49.03	48.42	48.76	49.72	50.00
Miscellaneous	0.20	0.13	0.14	0.15	0.12	0.17	0.15	0.13	0.14	0.14	0.13
Total	60.71	149.21	228.61	304.89	376.03	436.16	538.51	621.04	729.80	844.54	940.93

Table 1. Time (secs) to perform pentesting on projects up to 100 VMs. *Italicized* rows describe substeps of Mirror Creation.

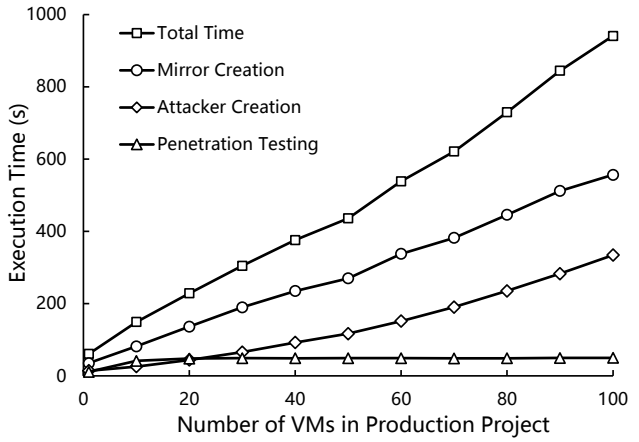

Figure 6. Time to perform pentesting on projects up to 100 VMs. The lines in the graph plot data from the major rows of Table 1.

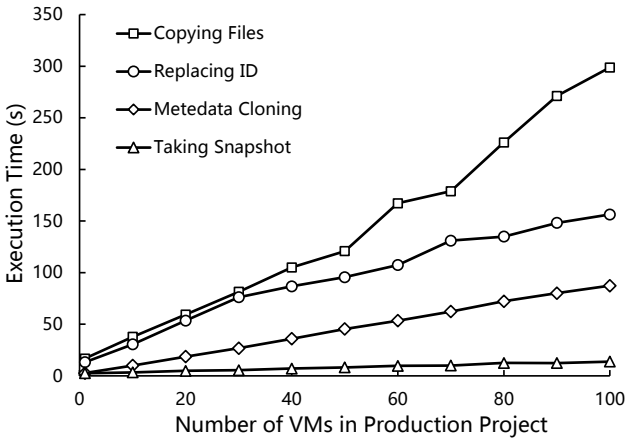

Figure 7. Time to perform the substeps of Mirror Creation for projects up to 100 VMs. The lines plot data from the italicized rows of Table 1.

standard OpenStack "controller node"; a second served as the standard "network node," hosting various network services; and a third hosted our POTASSIUM servers. The remaining four nodes were configured as "compute nodes"—nodes that host VM instances—running the Nova Compute service and our modified versions of QEMU and libvirt. Each compute node was equipped with four 2.2 GHz Intel Xeon E5-4620 8-core CPUs, 128 GB memory, one 250 GB SATA disk (7,200 rpm), and six 600 GB SAS disks (10K rpm). The network link speed between nodes was 1 Gbps.

Each compute node was configured to use three of its available disks. The 250 GB disk contained the node's root file system. One 600 GB disk contained the virtual disks of VMs created by OpenStack, and a second 600 GB disk contained the memory-state files of suspended VMs.

Finally, we divided the four computes nodes among two availability zones: two nodes for hosting production projects, and two nodes for hosting pentest projects.

4.1 Performance and Scalability of POTASSIUM

To measure the performance and scalability of our system, we created a series of eleven production projects with increasing size and measured the time it takes for POTASSIUM to complete external-mode pentesting. The sizes of the projects ranged from 1 to 100 VMs. For each project, we used a single disk image containing a vulnerable version of the WordPress blogging software to boot all of the project's VMs. Each VM is assigned a floating IP address, making it accessible from the external network and exposing the vulnerability to the outside world.

We used POTASSIUM to perform pentesting on each of our test projects and measured the time taken by the different steps within the overall process. For each project, POTASSIUM created one Attacker for every five nodes in the production project. Because the purpose of these experiments was to measure the "cloud-related actions" taken by POTASSIUM, the Attackers did not perform comprehensive testing; instead, we tailored them to look only for the WordPress vulnerability. In each run, POTASSIUM detected all of the vulnerable VMs.

The performance results of our end-to-end time trials are shown in Table 1, Figure 6, and Figure 7. The table shows that the absolute times suggest the practical feasibility for our pentesting as a service approach. For example, the total time for creating a pentest project associated with 30 VM production project is 940 seconds (less than 16 minutes), which is a reasonable time for such an operation. Further,

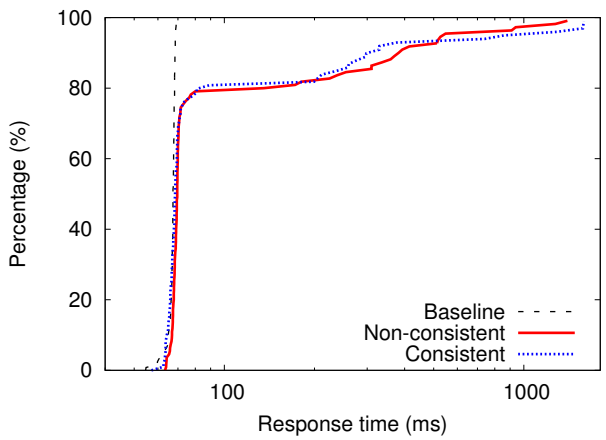

Figure 8. CDF of HTTP response time under different checkpointing scenarios. Note the log scale of the X-axis.

	Median	95%	Maximum	Average
Baseline	67.4	68.6	71.5	66.93
Non-consistent	69.6	526.7	1,397.8	152.49
Consistent	68.5	739.3	1,587.3	165.93

Table 2. Characterization of HTTP response times (in ms).

as Figure 6 illustrates, the total time grows linearly with the size of the production project; we conclude that our implementation shows practical scalability.

Within each POTASSIUM run, the most time-consuming step is the creation of the mirror subproject, and the time within that step is broken down in Figure 7. In that figure, one can see that copying the memory-state files of checkpointed VMs—from the production project to the pentest project, and thus across physical hosts in separate availability zones—accounts for the most time, and that time grows fastest as the size of the production project increases. This is because our cloud setup included only four compute nodes. Many VMs were therefore co-resident on our physical hosts, and consequently, the performance of memory-state image copying and ID replacement could be limited by available disk and network I/O bandwidth on our compute nodes. In a cloud with more physical hosts, we expect that these bottlenecks would be reduced.

In Table 1, the *Taking Snapshot* row shows the total time needed for the Snapshot Agents to complete their work. For any single VM in the production project, the pause time is much lower. (Recall from Section 3.2 that VMs in the production project are only unavailable during their pause times.) In our production project with 100 VMs, the pause time of each VM ranged from 88 ms to 757 ms, with an average of 300 ms.

4.2 Impact on the Production Project

We further measured the impact of live, consistent checkpointing on production projects by performing a series of experiments on a pair of communicating VMs within our OpenStack

Snapshot Type	Memory Snapshot	Disk Snapshot	Total Snapshot
Non-consistent	710.9	196.5	907.4
Consistent	692.5	203.6	896.1

Table 3. Average time required to checkpoint a VM (in ms).

setup. Each VM was a "small.m1" machine (2 VCPUs, 2 GB memory, 20 GB disk) running Ubuntu 14.04. One VM ran an Apache web server hosting WordPress; the other ran a MySQL database. We used Autobench [27] to measure the time it takes for this system to respond to HTTP requests under different situations as described below. We ran Autobench on a third VM and placed it on a different physical machine in order to simplify the test being performed. We wrote a script that repeatedly invokes Autobench to fetch the default WordPress "Hello World." Our script directs Autobench to send 10 requests in one second. Autobench waits for the replies and then terminates; our script then runs Autobench again.

We measured the HTTP response-time characteristics of our WordPress setup under three scenarios. In the first, we added no checkpointing activity to the system: this allowed us to obtain the baseline performance of the system. In the second, we subjected the VMs to periodic, live, *non-consistent* checkpointing activity. In the third, we subjected the VMs to periodic, live, *consistent* checkpointing as performed by POTASSIUM. In the second and third scenarios, we performed ten snapshots during the approximately 120 seconds of the test. This is an unrealistic checkpointing frequency for a POTASSIUM deployment: realistically, POTASSIUM would checkpoint a production project only rarely, once per pentest run. In our experiments, however, we performed frequent checkpointing in order to better understand the observable impact of checkpointing.

Figure 8 illustrates the HTTP response times of our system under the three test conditions. The graph shows that under normal (non-checkpointing) conditions, almost all requests complete within 70 ms. In both of the checkpointing scenarios, approximately 20% of the responses take longer than 70 ms. Table 2 shows the median, 95th percentile, maximum, and average response time in each of our three test scenarios.

To help explain these results, we measured the time taken by the various snapshotting steps for both non-consistent and consistent live checkpointing. By performing ten checkpoints under each algorithm, we obtained the memory-snapshot and disk-snapshot times shown in Table 3. The total snapshot time is about 900 ms, the fourth column in Table 3, for both cases but the actual pause time is only ≈220 ms, which includes the time it takes to save dirty memory pages for the last iteration and to take a disk snapshot. During the period a checkpoint is taken, the time to respond to a request can increase, since more load is present in the system (to save memory and disk states). Although checkpointing clearly has a performance impact, we conclude from our experiments that its impact is limited, because individual checkpoint times are short.

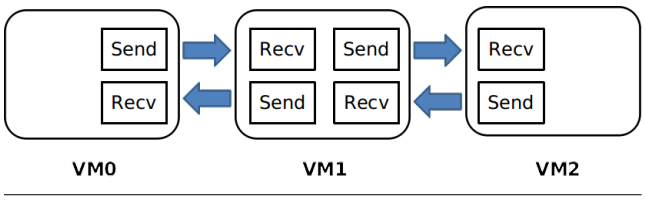

Figure 9. Consistent Snapshot Experiment

Our results also show that the performance impacts of non-consistent and consistent checkpointing are similar. Figure 8 shows that in our experiments, consistent checkpointing "outperformed" non-consistent checkpointing over a limited range: under consistent checkpointing, a larger fraction of the responses were received in under 400 ms. Although consistent checkpointing buffers packets (which could reduce packet losses in our experiment), we believe this counterintuitive result arises mostly from our test setup. Under consistent checkpointing, the "long tail" of the response-time curve causes our request generator to send fewer requests during our 120-second test, which reduces the server's overall workload.

4.3 Validation of Checkpoint Consistency

To evaluate our implementation of live, consistent checkpointing, we used a cluster of three VMs as illustrated in Figure 9. Each VM ran Ubuntu 12.04. We configured each VM with a different memory size—VM0 with 1 GB, VM1 with 2 GB, and VM2 with 3 GB—so that each would take a different amount of time to save its memory state during checkpointing. This causes the VMs to pause at different times, and thus creates the potential for *inconsistent packets* as defined in Section 3.2. As shown in Figure 9, VM1 communicates via UDP with both VM0 and VM2 in a bidirectional manner. Each of our UDP senders transmits about 600 packets per second.

We used this setup to measure lost packets and inconsistent packets, for normal (non-consistent) checkpointing and our consistent checkpointing implementation. Lost packets occur when a receiving VM is paused (*P1* in Figure 3), and inconsistent packets happen when a packet from a post-snapshot VM is delivered to a VM that has not completed its snapshot (*P2* in Figure 3). We present the results from one run of each checkpointing algorithm, in Table 4, with reference to the timeline of Figure 3 to illustrate events. Table 4 shows the numbers of lost and inconsistent packets that occurred during our experiments. Other runs show a similar distribution of lost and inconsistent packets.

Normal snapshot. A normal checkpoint causes about 130 packet losses for each UDP stream in the three-VM production project. This is in line with our observations that our UDP senders transmit about 600 packets per second and the average VM pause time is about 220 ms. In the mirror subproject created from the non-consistent snapshot, we saw a large number of inconsistent packets for streams *VM0→VM1* and *VM1→VM2*. Consider the *VM0→VM1* stream: VM0 resumed at time *t2* and continued sending packets to VM1. Because this is the normal snapshot case, the packets received between *t2* and *t4* affected the state of VM1's snapshot, although with respect to VM0's snapshot, they had not yet been sent. In other words, these packets cause an inconsistency between the VM0 and VM1 snapshots.

	Normal Snapshot		Consistent Snapshot	
UDP Stream	Prod. Project	Mirror	Prod. Project	Mirror
VM0→VM1	147	**527**	0	0
VM1→VM0	111	702	98	794
VM1→VM2	138	**428**	0	24
VM2→VM1	148	603	0	1,300

Table 4. Lost and inconsistent packets during normal (non-consistent) and consistent checkpointing. Plain-text numbers count lost packets; **boldface** counts inconsistent packets.

We also saw packet loss in the mirror subproject, for streams *VM1→VM0* and *VM2→VM1*. When VM1 is restored from the snapshot, it starts sending packets from the time at which it was paused (*t3*). However, when VM0 is restored from the snapshot, it is still expecting packets sent by VM1 about a second earlier, at *t1*. A similar situation occurs for the stream *VM2→VM1*.

Consistent snapshot. Under consistent checkpointing, in the production system, we saw packet loss only in the *VM1→VM0* stream. With reference to Figure 3, these are the *P1* packets that are dropped while VM0 is paused between *t1* and *t2*. Once VM0 finishes its snapshot, it starts to send red packets. Once VM1 receives a red packet, it buffers all subsequent packets until it completes its snapshot. (Even when a VM is paused during live checkpointing, the network driver can still buffer incoming packets for that VM.) Thus, no packet losses happen on VM1 or VM2. VM1 buffered 2,370 packets during *t2–t4* and VM2 buffered 785 packets during *t4–t5*.

In the mirror subproject created from the consistent snapshot, we did not see any inconsistent packets. This validates our implementations of packet coloring and buffering for creating consistent snapshots.

In the stream *VM2→VM1*, we saw 1,300 lost packets. Those occurred because once VM0 starts to send red packets at *t2*, VM1 starts to buffer packets: this includes the packets sent by VM2. While these packets are buffered and eventually delivered to VM1 in the production system, they are effectively lost to the snapshot of VM1 that will be resumed in the mirror system.

We saw 24 packets lost in *VM1→VM2*. That could be because VM1 was restarted a little earlier than VM2.

4.4 Case Study 1: Pentesting a LAMP Stack

We tested our POTASSIUM implementation by using it to pentest a small but representative cloud server deployment. LAMP (Linux, Apache, MySQL, and PHP) is one of the most popular platforms for deploying services within clouds, and

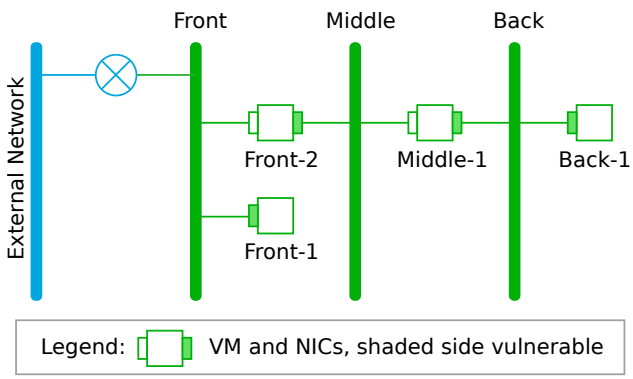

Figure 10. Project setup for evaluating pivot-mode pentesting. Only the shaded interfaces are vulnerable to attack.

WordPress is one of the most common content-management systems built atop MySQL and PHP. We used this software to validate POTASSIUM's ability to discover vulnerabilities in cloud-hosted systems.

Metasploit includes 34 (and counting) exploits for various configurations of WordPress. We focused on a WordPress plugin with a vulnerability that was disclosed on December 3, 2014 [28]; the Metasploit module that exploits this vulnerability was published on December 12. (The Metasploit module is called `wp_downloadmanager_upload`.) The vulnerability allows uploading arbitrary files to the server by sending carefully constructed HTTP requests.

We created a production project to host the vulnerable configuration of WordPress within in our OpenStack environment. We then invoked POTASSIUM to perform pentesting, using its various pentesting modes (Section 3.3). From the configuration of the production project, we could predict which vulnerabilities should be exposed to which Attackers within the pentest project. For example, in external mode, only the "first layer" of the pentest project's topology was exposed, mimicking an outsider's view of the project. POTASSIUM correctly detected the expected vulnerabilities in all test cases.

4.5 Case Study 2: Pivot-Mode Pentesting

We performed a second case study to highlight the effectiveness of pivot-mode pentesting. We built a production project with a carefully distributed set of vulnerabilities and a known, potential "chain" of compromises. We used POTASSIUM to perform pentesting against this project to determine whether it could find the compromise chains.

The network topology and the vulnerability distribution for this case study are shown in Figure 10. The topology included three networks with a layered structure: *Front*, *Middle*, and *Back*. Each of the four VMs ran a vulnerable configuration of WordPress atop the LAMP stack. Two of the VMs were equipped with two NICs and connect to two networks, but on these nodes, the Apache web server listened on only one interface: thus, those nodes were vulnerable only on a single interface. The *Front-2* VM was vulnerable only from the *Middle* network, and the *Middle-1* VM was vulnerable only from the *Back* network. Per pivot-mode pentesting, once a machine is exploited, POTASSIUM assumes that the attacker can gain full control of the system and be able to start the next round of attacking.

Stage	Time
Mirror Creation	227.59
Attacker Creation	77.56
Penetration Testing	35.99
Miscellaneous	0.87
Total	342.01

Table 5. Time (secs) to perform pivot-mode pentesting on (multiple clones of) the project shown in Figure 10.

VM	Directly Exploited	Indirectly Exploited
Front-1		
Front-2	Front-1	
Middle-1	Front-2, Back-1	Front-1
Back-1	Middle-1	Front-2, Front-1

Table 6. Compromise chains found by pivot-mode pentesting. From *Middle-1* or *Back-1*, all the other VMs can be exploited.

We used our POTASSIUM implementation to perform pivot-mode pentesting on this project. As detailed in Section 3, the process includes mirror subproject creation, attack subproject creation, and penetration testing. The time taken for each stage in this case study is shown in Table 5. Our implementation of pivot-mode pentesting creates multiple pentest projects at once so that it can run the pentesting rounds in parallel. However, only actual pentesting is performed in parallel: the multiple pentest projects are created one by one. Thus, most of the time in this case study is consumed by mirror and attacker creation.

The compromise chains discovered by POTASSIUM are shown in Table 6. The results correspond to what one would expect from an analysis of Figure 10. We therefore conclude that our implementation of pivot-mode pentesting is able to perform "what-if" analyses and detect possible compromise scenarios in a cloud project.

5. Related Work

As a cloud service, POTASSIUM is related to a variety of proposed "as-a-service" cloud offerings [1, 4, 5, 13, 14, 16, 23, 24, 34, 38]. Of these, IT-as-a-service [24] and Disaster-Recovery-as-a-service [38] are conceptually most related to our work. Like POTASSIUM, they deal with the use of cloud computing to facilitate tasks performed by IT professionals. The panel discussion summarized in [24] deals with IT tasks in general, while using the cloud to efficiently realize disaster recovery is proposed in [38]. To our knowledge, however, our work is the first to propose the use of cloud computing to realize penetration testing as a cloud service.

Given the importance of pentesting as a tool for IT security professionals, it has been explored along several directions that relate to our work. In Dan Lambright's invited talk presented at USENIX LISA 2014 [22], he recognized the inherent tension between cloud tenants who want to run pentests on their system and cloud providers who do not want to run the risk of harming their infrastructure or tenants. He suggested that cloud tenants could use Docker containers [26] to simulate attacks on their cloud resources. To solve the scalability issues with penetration testing in large and complicated systems, like cloud platforms, Bugcrowd proposed crowd-sourcing pentests on demand [9, 37]. Commercial pentesting services of cloud resources are available. For example, the "CloudInspect" service allows pentesting of AWS-hosted VMs [11]. From the company's web site, cloud users are able to log into their AWS account and order pentesting services against their running (production) cloud resources. As a third party, there are certain restrictions on the types of VMs they are able to pentest [11].

Previous efforts have proposed the automation of the actual pentesting process in a manner similar to our approach in POTASSIUM. Jianbin et al. [15] proposed an architecture that consists of pentest managers and executors, similar to the POTASSIUM Coordinator and Attackers. Kamongi et al. [19] proposed a system, called Nemesis, for finding and evaluating vulnerabilities in the cloud. Building on their own earlier work [20], the Nemesis system interacts with a database of vulnerabilities and existing knowledge about the production system. Like POTASSIUM, it makes use of Metasploit to execute the potential exploits.

The key difference between our work and these earlier pentesting efforts is that POTASSIUM performs the automated pentesting on a cloned version of the production system. As we have shown, this property is critical to reduce the impact on the production system and to allow more extensive and potentially invasive tests to be performed.

The consistent checkpointing approach we use in POTASSIUM is inspired by several earlier efforts on building protocols for taking consistent distributed snapshots. Kangarlou et al. [21] proposed to use live migration to realize live snapshots and they implemented a message coloring approach in routers to realize consistent snapshots. In their approach, however, they do not buffer packets that could lead to inconsistent snapshots, but simply drop them. In hotSnap, Cui et al. [12] designed a mechanism to rapidly take a snapshot by using a Copy-On-Write approach for saving both the memory and disk state. They also implemented packet coloring and buffer packets that could lead to inconsistent snapshots. Our consistent distributed snapshot implementation is based on the hotSnap system design.

Because of the multi-tenant nature of the cloud, physical resources are shared by multiple users and this can lead to interference and unpredictable performance. Schad et al. [33] observed that the performance of CPU, memory speed, I/O and network bandwidth in Amazon EC2 is at least an order of magnitude *less stable* than an equivalent physical cluster. At different times of a day, the performance of the AWS also changes [29]. The "noisy-neighbor" problem [25] has been discussed in previous work when it comes to VM co-residence. Many solutions have been proposed, for instance Angel et al. proposed the Pulsar system to ensure throughput guarantees across all resources [3]. While these approaches could be applied to our work, we opted for a pragmatic but effective solution in realizing POTASSIUM. Specifically, to avoid performance interference with the production system, all virtual machines associated with pentesting are placed in a different cloud availability zone, which does not share any physical resources with the production system.

6. Conclusion

In this paper, we proposed penetration testing as a service (PTaaS) to enhance security for cloud platforms. We developed a PTaaS architecture, called POTASSIUM, and prototyped it by extending the OpenStack cloud environment. Specifically, we realized POTASSIUM by extending OpenStack with project mirroring, a live, consistent checkpointing mechanism, and an automated pentesting module. By taking a live snapshot of a production system, POTASSIUM captures the exact state of the running system into a mirror subproject. By running penetration testing against this mirror subproject, the impact on the production system can be minimized. In our evaluation, we showed that our system is practical, can successfully detect real-world vulnerabilities, and that the impact on the production system is limited.

Acknowledgments

We thank our University of Utah colleagues Dan Bowden, Steve Corbató, and Jake Johansen of University Information Technology for consulting with us to share their expertise on pentesting production systems. We also thank our shepherd, Ymir Vigfusson, and the anonymous reviewers for their valuable comments and help in improving this paper. This material is based upon work supported by the National Science Foundation under Grant No. 1314945.

References

[1] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally. Database management as a service: Challenges and opportunities. In *Proc. ICDE*, pages 1709–1716, Mar.–Apr. 2009.

[2] Amazon Web Services, Inc. AWS penetration testing. http://aws.amazon.com/security/penetration-testing/, 2015.

[3] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *Proc. OSDI*, pages 233–248, Oct. 2014.

[4] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: Schema-mapping techniques. In *Proc. SIGMOD*, pages 1195–1206, June 2008.

[5] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. A comparison of flexible schemas for software as a service. In *Proc. SIGMOD*, pages 881–888, June–July 2009.

[6] F. Bellard and the QEMU Team. QEMU. http://www.qemu.org/.

[7] S. M. Bellovin and R. Bush. Configuration management and security. *IEEE JSAC*, 27(3):268–274, Apr. 2009.

[8] A. Bryson. Penetration testing in the cloud. http://blogs.cisco.com/security/penetration-testing-in-the-cloud, Aug. 2011.

[9] Bugcrowd Inc. About Bugcrowd. https://bugcrowd.com/about, 2015.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. NSDI*, pages 273–286, May 2005.

[11] Core Security Technologies. Core Security offers on-demand cloud security testing service for Amazon Web Services. http://www.coresecurity.com/content/Core-Security-Offers-On-Demand-Cloud-Security-Testing-Service-for-Amazon-Web-Services, June 2011.

[12] L. Cui, B. Li, Y. Zhang, and J. Li. HotSnap: A hot distributed snapshot system for virtual machine cluster. In *Proc. LISA*, pages 59–74, Nov. 2013.

[13] W. Dawoud, I. Takouna, and C. Meinel. Infrastructure as a service security: Challenges and solutions. In *Proc. 7th International Conference on Informatics and Systems (INFOS)*, Mar. 2010.

[14] H. Hacıgümüş, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proc. ICDE*, pages 29–38, Feb.–Mar. 2002.

[15] J. Hu, Y. Wang, C. Tang, Z. Guan, F. Ren, and Z. Chen. A novel framework to carry out cloud penetration test. *International Journal of Computer Network and Information Security*, 3(3):1–7, Apr. 2011.

[16] W. Itani, A. Kayssi, and A. Chehab. Privacy as a service: Privacy-aware data storage and processing in cloud computing architectures. In *Proc. DASC*, pages 711–716, Dec. 2009.

[17] J. Johansen. Personal communication, Apr. 2015.

[18] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen. Analysis of vulnerabilities in Internet firewalls. *Computers & Security*, 22(3):214–232, Apr. 2003.

[19] P. Kamongi, M. Gomathisankaran, and K. Kavi. Nemesis: Automated architecture for threat modeling and risk assessment for cloud computing. In *Proc. 6th ASE International Conference on Privacy, Security, Risk and Trust (PASSAT)*, Dec. 2014.

[20] P. Kamongi, S. Kotikela, K. Kavi, M. Gomathisankaran, and A. Singhal. Vulcan: Vulnerability assessment framework for cloud computing. In *Proc. SERE*, pages 218–226, June 2013.

[21] A. Kangarlou, P. Eugster, and D. Xu. VNsnap: Taking snapshots of virtual networked environments with minimal downtime. In *Proc. DSN*, pages 524–533, June–July 2009.

[22] D. Lambright. Penetration testing in the cloud. https://www.usenix.org/conference/lisa14/conference-program/presentation/lambright, Nov. 2014. Invited talk at LISA '14.

[23] G. Lawton. Developing software online with platform-as-a-service technology. *Computer*, 41(6):13–15, June 2008.

[24] G. Lin, D. Fu, J. Zhu, and G. Dasmalchi. Cloud Computing: IT as a Service. *IT Professional*, 11(2):10–13, Mar./Apr. 2009.

[25] M. Lodge. Getting rid of noisy neighbors: Enterprise class cloud performance and predictability. http://blogs.vmware.com/rethinkit/2010/09/getting-rid-of-noisy-cloud-neighbors.html, Sept. 2010.

[26] D. Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.*, Mar. 2014.

[27] J. T. J. Midgley. Autobench. http://www.xenoclast.org/autobench/, Mar. 2012.

[28] M. Nadeau. Security advisory – high severity – WordPress download manager. http://blog.sucuri.net/2014/12/security-advisory-high-severity-wordpress-download-manager.html, Dec. 2014.

[29] S. J. Nadgowda and R. Sion. Cloud performance benchmark series: Amazon Elastic Block Store (EBS), Amazon Simple Storage Service (S3), and Amazon EC2 Instance Local Storage. Technical report, Stony Brook Network Security and Applied Cryptography Lab, Oct. 2010. http://digitalpiglet.org/research/sion2010cloud-storage.pdf.

[30] OpenStack Foundation. OpenStack Juno. https://www.openstack.org/software/juno/, Oct. 2014.

[31] E. Ramirez-Silva and M. Dacier. Empirical study of the impact of Metasploit-related attacks in 4 years of attack traces. In *Advances in Computer Science – ASIAN 2007: Computer and Network Security*, volume 4846 of *LNCS*, pages 198–211. Springer, 2007.

[32] Rapid7, Inc. The Metasploit Framework, version 2015031001. https://github.com/rapid7/metasploit-framework/releases/tag/2015031001, Mar. 2015.

[33] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. In *Proc. VLDB*, pages 460–471, Sept. 2010.

[34] H. E. Schaffer. X as a service, cloud computing, and the need for good judgment. *IT Professional*, 11(5):4–5, Sept./Oct. 2009.

[35] D. Veillard. libvirt: The virtualization API. http://libvirt.org/.

[36] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

[37] D. Winder. Crowdsourcing security exploits: how the cloud can keep us safe. http://www.cloudpro.co.uk/cloud-essentials/cloud-security/5226/crowdsourcing-security-exploits-how-cloud-can-keep-us-safe, Jan. 2013.

[38] T. Wood, E. Cecchet, K. K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *Proc. HotCloud*, June 2010.

[39] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, June 2004.