

HY558 Report

Detecting Targeted Attacks Using Shadow Honey pots

This paper suggest a novel intrusion detection system called shadow Honey pots. Shadow Honey pots combine features from Honey pots and anomaly detection systems.

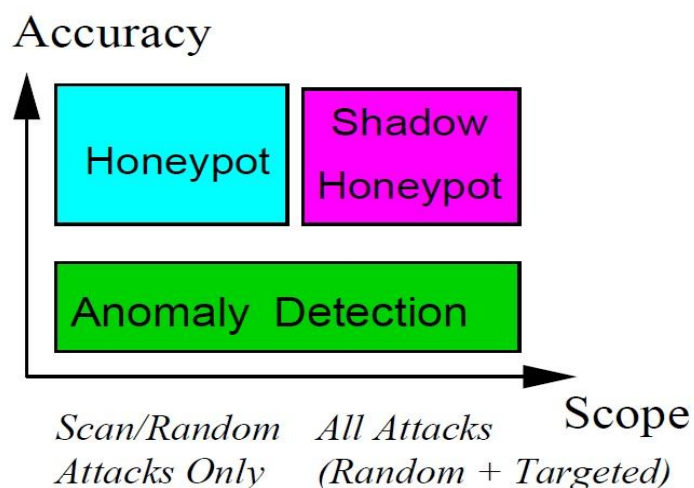
Currently there are two main types of detection systems:

1. Anomaly detection Systems.

This systems can detect a wide range of attacks but their efficiency bares the cost of false positives(e.g. Abstract Payload Execution).

2. Honey pots

Honey pots are systems that are (heavily) instrumented to detect potential attacks. They are very accurate but have a smaller scope.



The shadow Honey pots combine Honey pots with anomaly detections, in an effort to achieve high accuracy for variety of attacks. Shadow Honey pots can also be configured to detect client-side attacks. They are also easy to integrate and allow for fine-tuning against false positives.

Shadow Honey pots are cloned instances of the original software. These are instrumented to detect attacks and share state with the original software.

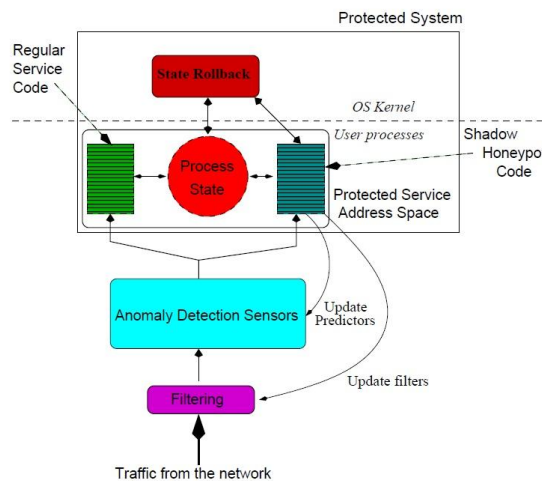


Figure 2: Shadow Honeypot architecture.

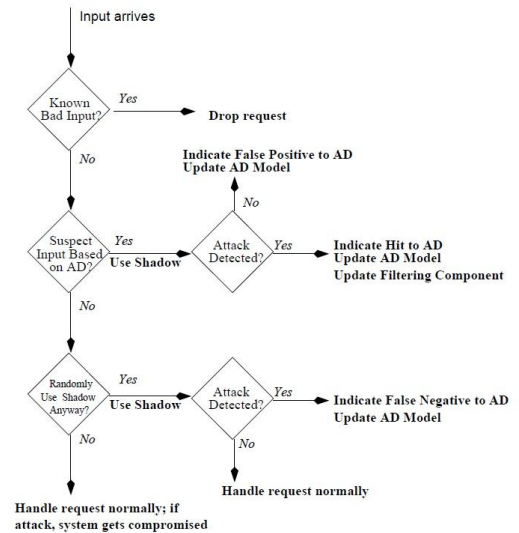


Figure 3: System workflow.

The system architecture is described in figures 2 and 3.

Requests first pass through *filtering* where if there are parts of known attacks they are blocked. After that, we have the *Anomaly Detection Sensors*. This part decides if the traffic consists of a “potential” attack (remember anomaly detection offers false positives). If an attack is considered potentially malicious, it is forwarded to the shadow honeypot. The shadow honeypot, in turn, is instrumented and can detect potential attacks. If it detects an actual attack, it rolls back the shared process state.

A sample system has been implemented and is described. This system is designed to detect buffer-overflow attacks. In order to create a shadow version of a system, we instrument at the source level (C-to-C transformations). Mainly, what we do is transfer all the buffers to the heap and allocate them using a custom *malloc()* function called *pmalloc()*. This function rounds up allocations at page boundaries. At every returned buffer, the function places write-protected pages before and after. These pages are simply TLB entries in order to reduce wasted space.

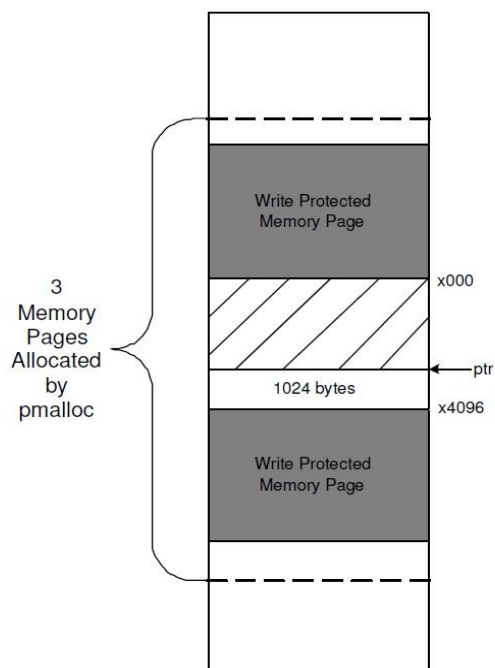


Figure 5: Example of *pmalloc()*-based memory allocation: the trailer and edge regions (above and below the write-protected pages) indicate “waste” memory. This is needed to ensure that *mprotect()* is applied on complete memory pages.

In order for the system to have the ability to roll back, a new system call has been added called transaction. Assuming we deal with transactional applications we need to place this system call at three points: (i) before the processing of a request, (ii) after the processing of a request, (iii) inside the signal handler(SIGSEV).

This however is one of the weaknesses of the system since we rely on the programmer, to place the system call correctly. The system also when loosely coupled for client side attacks, fails to deliver, since it cannot emulate user behavior. Another weakness is that an attacker can produce specific false positives to soften the anomaly detectors, and then attack.