5-2021

Architectural Style: Distortions for Deploying and Managing Deception Technologies in Software Systems

Elijah Cantella
edc8230@rit.edu

R·I·T

Architectural Style: Distortions for Deploying and Managing Deception Technologies in Software Systems

by

Elijah Cantella

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Software Engineering

Department of Software Engineering
Golisano College of Computing and Information Sciences

Rochester Institute of Technology
Rochester, NY

May 2021

The thesis “Architectural Style: Distortions for Deploying and Managing Deception Technologies in Software Systems” by Elijah Cantella has been examined and approved by the following Examination Committee:

Kal Rabb — Date
SE Faculty
Thesis Advisor

Dr. Naveen Sharma — Date
Department Chair
Thesis Committee Member

Dr. J Scott Hawker — Date
Graduate Program Director
Thesis Committee Member

Abstract

Deception technologies are software tools that simulate/dissimulate information as security measures in software systems. Such tools can help prevent, detect, and correct security threats in the systems they are integrated with. Despite the continued existence and use of these technologies (~20+ years) the process for integrating them into software systems remains undocumented. This is due to deception technologies varying greatly from one another in a number of different ways. To begin the process of documentation, I have proposed an architectural style that describes one possible way deception technologies may be integrated into software systems.

To develop this architectural style, I performed a literature review on deception technologies and the art of deception as a discipline. I break down how deception technologies work according to the art of deception through the simulation and dissimulation of software components. I then examined existing deception technologies and categorize them according to their simulations/dissimulations. The documented and proposed architectural style describes how software systems deploy and manage deceptions. Afterwards, I propose a number of future research opportunities surrounding this subject.

Contents

1 Introduction

In an ever-growing digital landscape software attackers and defenders are constantly trading blows. It is not so much a question of whether or not any vulnerabilities do exist in a piece of software, but of what/how many vulnerabilities exist. Vulnerabilities are discovered and exploited daily. Accenture, a business and technology consultant, estimated in 2019 that there are 5 trillion USD at risk of cybercrime over the next 5 years [10]. If companies fail to meet and exceed safety standards for their software products, they are liable to become victims of cybercrime. There are many techniques and solutions that help make software more secure, ranging from simple design or coding practices to the integration of advanced network protocols. The best protection is often the result of combining many solutions and practices together. One practice that may bolster existing solutions to provide a much-needed edge for software security are deception technologies (DTs).

DTs are software tools that apply techniques to divert or confuse malware as a security mechanism in software systems. The specific purpose for each technique varies greatly, even within the same category of DT. One category of DTs called honeypots "… [are a] security resource whose value lies in being probed, attacked, or compromised" [2]. This kind of DT often employs the technique of simulating false systems or services in hopes of engaging with malware or malicious users. In this category alone it is possible to show how vastly different one DT is from another.

Take 2 DTs from the category of honeypots and compare them with one another. For this example, examine the Deception Toolkit (DTK) and Capture-HPC [6]. DTK, one of the first publicly available DTs, is described as a low-interaction server honeypot that mocks services hosted on Unix systems. This DT slows down malicious users by obfuscating whether or not attacks on hosted services are successful. Capture-HPC on the other hand is described as a high-interaction client honeypot that monitors changes in the state of the system (files, registries, and processes). This DT is more suited for detecting malicious servers, and when triggered it sends notifications to the developers. This DT could serve as a decoy when integrated with a software system, engaging with potentially dangerous servers to verify that it is safe for the software system to interact with the desired servers. The common unifying factor between these two DTs, and all DTs for that matter, is that they hide true information and/or show false information for

some kind of security benefit. The process of hiding and showing information is called distortion, and is a core element in the “art of deception” (which is the study and practice of deception).

While these technologies may prove beneficial for many software systems, there are a number of things to consider when working with them. Xiao Han et al. [5] for example identified that it is unknown how DTs are deployed and managed in specific software systems. They also noted the vast number of ways that DTs may be categorized along with other research groups [4, 5, 6]. Some of the ways DTs may be categorized include their purpose for being, their method of deployment, the layer in the software stack they take place, the level of interaction they involve, etc.

It is because of these great variances in traits and intentions that there are no documented architectures dedicated to the integration of DTs. The closest thing to a documented architecture as a published work involves a general engineering process for integrating the art of deception into a software system [3]. This published paper from 2014, to the best of my knowledge, is the closest work to describing the architecture of such a system. It is lacking, however, in that it fails to fully describe what a system would look like if it underwent such a process. It is for this reason I return to the art of deception and study DTs; to propose a new architectural style for integrating DTs into software systems.

2 Research Objectives

The primary objective of this study is to propose an architectural style (referred to simply as style from this point on) that reflects an emphasis on integrating DTs into software systems. This section captures the scope of the thesis.

2.1 Documentation

The documentation of the style is a critical element necessary to complete the objective. As such, research was conducted to elicit the required artifacts to fully depict the proposed style. The documentation adheres to the "Gang of Four" styles guide with slight modifications where necessary.

As for the subject of the documentation, I decided to document an architectural style over an architectural pattern. I have defined the difference between the two as the following:

> **Architectural Style:** A conceptual way of viewing, organizing, and developing a system (ex: client-server).
>
> **Architectural Pattern:** A concrete way to implement an architectural style. Solves issues raised by the overarching architectural styles (ex: proxy).

I made this distinction to begin with the highest level of abstraction. Once the style has been established, it will be possible to research the ways that style manifests as different architectural patterns.

More often than not architectures reflect the requirements and constraints of their systems. When a DT is introduced to a system it may add additional surface area, features, constraints, etc. Both the underlying system that integrates the DT and the DT itself should be designed in such a way to maintain robustness with as little complexity as possible.

There is a back-and-forth relationship between the design of DTs and the systems that integrate them. To help understand this relationship consider that a DT is like a tool and a software system is like a construction project. Tools are designed to fulfill a function, and do not care about when they are used. They do, however, care about how they are used. There is a right way to use a tool, and a wrong way to use a tool. Tools are used in construction projects as mechanisms to complete tasks. Very similarly, DTs are used in software systems as security mechanisms. So,

while it is possible to create a new DT that fulfills a specific function, it is still necessary for a system to understand that DT and how to use it. It is because of this I place the responsibility for accommodating the other technology on the system and not the DT. The proposed style therefore describes how a software system uses DTs.

2.2 Theoretical Solution

I have conducted research into relevant published works and identified critical documents that contribute to the styles surrounding systems that integrate DTs. For more information on these works, see *Section 3*. In addition to the published works, I examine a number of DTs hosted on GitHub to consider their technical requirements for proper integration.

The style is developed with the art of deception and technical aspects in mind. Both are necessary to fully comprehend DTs. A concrete DT is a piece of software where the art of deception has been applied. A user must understand the art of deception to understand the function of the DT, additionally, a user must understand the technical aspects to better execute that function. The proposed style is therefore strictly a way to view a system that integrates DTs. It does not necessarily reflect the styles being used in the cybersecurity domain. Rather, it is based on the art of deception and the anticipated practices that developers should undergo, considering the technical aspects.

I do not test to see whether or not the proposed style is prevalent as a current practice. It may exist as proposed, exist with variations, or not exist at all. From this proposed style a system may be built from scratch to test its applicability or research may be conducted to discover the actual styles that exist in practice.

Another important note is that the proposed style is meant to describe DTs being integrated into production systems over research ones. The distinction being that a production system uses DTs as a security mechanism while a research system does not; a production system must protect its CIA (confidentiality, integrity, and availability; 3 core principles of software security) while a research system may compromise its CIA.

3 Related Works

The surrounding literature comprises a mixture of published materials that range from the social sciences (specifically with the art of deception) to surveys of software implementations. For the purpose of documenting the style it is necessary to fully comprehend the nature of DTs, which begins with understanding how deception works.

The first set of published papers consist of the theory of deception [1, 8, 9]. Bell and Whaley published a book that covers the art of deception [1]. They draw upon a number of historical events and other research venues to propose a framework for deception, which may be applied to any field. Eventually after discussing the theory of deception, papers begin to apply them. Flowing from this practice, studies have been conducted to determine how to make deception more effective. A social experiment, for example, sought the best way to encourage a passerby to engage in an activity [9]. Eventually this practice leads into software. Game theory, for example, can be applied to honeypots to determine an attacker's or defender's most desirable action when provided with a given situation [8].

One of the first published works documenting deception in software was by Lance Spitzner [2]. Spitzner observed the practices taking place at the time and documented them in a cohesive way by creating his definition of both deception and honeypots. While some works expand on one specific kind of DT as "Honeypots: Tacking Hackers" [2] did, others study and breakdown multiple kinds of DTs through surveys [4, 5, 6]. These surveys usually examine concrete implementations and describe the technologies in various ways. It is within this set of published works that I recognized a need to document styles for software systems that integrate deception.

The closest related work that describes the integration of DTs into software systems was conducted in 2014 [3]. To my knowledge this is the only paper that attempts to mainstream the process for integrating DTs into software systems. The paper describes the process a developer should undergo when integrating a DT. While this is on the right track it could more clearly describe the architecture of a system that undergoes such a process.

4 Methodology

The goal of this thesis is to document a new style for integrating DTs into software systems. This is done by bridging the gap between the art of deception and the technical aspects of using a DT. The following section outlies the steps taken during the thesis.

4.1 Research Questions

There were three research components necessary to propose the style. I considered the developers' concerns, the technical requirements for integration, and the documentation process. I have formulated each component as a research question (RQ) that was addressed by the methodology. Each RQ and its rationality upon conception are described below.

RQ1: What concerns should developers have when integrating deception technologies into any software system?

Developers and cybersecurity practitioners design and implement software systems that use DTs, and when they do so they consider a number of values and constraints for their systems. Amongst the things they consider are the deception they wish to deploy, and the state of the system it is deployed to. I should also consider these values and constraints. While I may not arrive at the same style, it will be more applicable if I'm able to adhere to their values and standards.

To gauge this RQ I undergo a literature review regarding the concerns, values, and practices of DT users. I have also studied the art of deception to better understand its practice.

RQ2: How does the architectural style accommodate the different kinds of deception technologies?

There are several kinds of DTs whose design and implementation vary greatly. I study each specific kind of DT and consider how the documented style will accommodate it.

Because of this a literature review is conducted on the taxonomies of DTs. After this specific literature review, the relationships between the specific technologies and the styles are considered.

RQ3: What documents and diagrams are necessary to accurately depict the architectural styles?

After discovering the style, I must document it appropriately so it can be communicated with ease. The contribution of this thesis will be drastically limited if the style is developed but not effectively documented. There should be as much documentation as required, no more and no less, to fully depict the style.

4.2 Accepted Frameworks

My literature review in *Section 3* provided me with a number of different frameworks and taxonomies to work with. I chose the following frameworks and terminology for conducting the thesis.

4.2.1 Art of Deception

For the art of deception, I chose to accept the framework documented by Bell and Whaley [1]. After reading their work and a quick application of their framework I understood how it was applicable to DTs, several other papers I reference also cite their work [3, 4]. They describe deception as distorting reality. This is done my dissimulation (hiding what is real) and simulation (showing what is false). There are 6 types of simulations/dissimulations that can be used to hide and show information, these are: masking, packaging, dazzling, mimicking, inventing, and decoying. When these pieces of information come together, they are used to create a ruse. A ruse generally falls into 1 of the 5 categories: unnoticed, benign, desirable, unappealing, and dangerous. When a ruse is used appropriately it yields some sort of effect or cover. For more information on this framework, I recommend reading the book “Cheating and Deception” [1].

4.2.2 Types of DTs

For the various types of DTs, I accepted the following as described by Fraunholz et al. [4]: ASLR (address space layout randomization), steganography, MTD (moving target defense), honeypots, honeyports, and honeytokens. All these technologies are considered DTs by the researchers because it falls into one of the 9 categories described by Dunnigan and Nofi [12]. “Their taxonomy differentiates nine classes of deception: concealment, camouflage, false and planted information, lies, displays, ruses, demonstrations, feints and insight” [4]. All of these 9 categories simulate/dissimulate information, though they are specific categories of deception. It is based on these 9 classes that Fraunholz et al. were able to categorize the above technologies as DTs.

Not only are the types of DTs because of the classes of deception, but out of the examined surveys these DTs appeared to be the most common. This is not an exhaustive list of all DTs. All the DTs I selected (*Appendix A*) fall under one of these categories and allows for information to either be simulated or dissimulated.

4.2.3 Deployment Methods of DTs

Another important category I thought necessary to gather information on DTs was the method of deployment to a system. For this I used the same categories as laid out by Xiao Han et al [5]. The DT is either built-in (placed in the system during the design phase), added-to, set in-front of, or isolated from the target system.

4.2.4 Software Characteristics

In addition to the frameworks previously described I specified the purpose of security relative to the system and the software levels that make up that system. The purpose of a security measure is to protect the confidentiality, integrity, and availability (CIA) of a system. The observed software layers include the system, data (sometimes referred to as application), and network layers.

4.3 Data Collection & Analysis

Data was collected and categorized from public GitHub repositories. I attempted to select at least 3 if not 5 DTs of each type, though I was unable to find 3 MTD technologies nor was I able to find 3 honeyport technologies. These repositories were found via key word searches and forum posts. The set of repositories and collected information may be viewed in *Appendix A*.

While I selected and examined these 20 technologies, I recognize that I could have re-examined DTs from datasets described in other surveys. I chose to create my own dataset and performed my own categorizations. This is something I have identified in *Section 6* which talks about threats to validity and future works.

4.4 Style Development

The style began by depicting a DT embedded in a software system. This was the clear starting point as it was the most fundamental requirement: a DT is integrated into a software system. The proposed style was developed beyond this point by taking steps as directed by the art of deception and the technical aspects of DTs.

As the style was developed, I documented each step and why it was taken. Each step was identified by examining the art of deception and the technical aspects of DTs. I created a diagram for each step and described the reasoning behind it.

For this paper adhered to the "Gang of Four" design documentation [11]. The design documentation specified the following features:

- Pattern Name
- Intent
- ~~Also Known As~~ (only one name is proposed)
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- ~~Sample Code and Usage~~ (language specifics are irrelevant to the style)
- ~~Known Uses~~ (how the style manifests as architectural patterns is outside of scope)
- Related Patterns

4.5 Proposal

After the style was developed, I submitted it to the members of the thesis committee. I documented any places that they found confusing and updated the style as they requested. Any changes made in this way were documented along with their rational in the timeline (*Appendix B*). This thesis proposes the style *(Section 5.5*) after the adjustments were made as requested by the committee.

5 Research Results and Evaluation

5.1 Approach

5.1.1 Understanding Deception in Software

As a precursor, the art of deception must be understood. I have read a number of books on this topic including “The Art of War” by Sun Tzu and “Cheating and Deception” by Whaley and Bell [1]. The previous book is referenced quite often amongst other published works regarding DTs and I have also chosen to work with their proposed framework. I shall briefly explain their structure of deception.

Deception involves distorting reality, the result of which is called an illusion. This is done through “dissimulation” (hiding what is real) and “simulation” (showing what is false) of information. The book described information as a characteristic that engages with a person’s senses, though for software I define information as an arbitrarily granular piece of software (that is as small as a bit or as large as an application if necessary). Masking, repackaging, and dazzling are forms of dissimulation, while mimicking, inventing, and decoying are forms of simulation. Many simulations/dissimulations work together to create ruses, which contributes to the illusion. A ruse generally exists for one of the following, to make something go unnoticed, to make something seem insignificant, to make something desirable, to make something unappealing, or to make something appear dangerous.

With this basic understanding it is possible to classify the selected DTs from out dataset and document the information they simulate/dissimulate, the ruses they attempt to create, and therefore the overall illusion. It is also possible to better comprehend the technical aspects of DTs as they relate to the art of deception.

The art of deception and the technical aspects of DTs meet at the characteristic level. A number of DT surveys document the concerns of developers for managing DTs after they are deployed [2, 5, 6]. Fingerprinting, or the idea that a DT can be discovered, bares a number of concerns. Not only is the illusion no longer as effective, the DT may add additional surface area to be exploited. It is for this reason and the general belief that deception is not an alternative to normal security conventions that I recommend that systems be allowed to exist without any deception.

Additionally, after examining DTs more thoroughly I found a point of contention between the realm of software and the art of deception (see RQ1).

5.1.2 How Deception Technologies Work

This basic framework can also be used to make new DTs or to make them more effective. One of the DTs I examined can operate technically on its own, but it won't work according to the art of deception. The DT is called "honeybits" and it spreads "breadcrumbs" around a system to point in the direction of a honeypot, a separate DT. These breadcrumbs consist of files, browser histories, and registry keys. On its own it won't accomplish much, but it can contribute to an existing illusion, i.e., a honeypot.

This was, however, the exception amongst the DTs I selected. Most of the DTs in my dataset can operate on their own when integrated into a system, both technically and via the art of deception. Each DT can be seen to simulate or dissimulate information (*Appendix A*). Additionally, it appears that there is no shortage of assets that can be simulated or dissimulated. It is at this point that I believe that DTs can be framed in such a way as described by Bell and Whaley.

First with the idea of hiding what is real and showing what is false, a software system that integrates DT consists of true and false assets. False assets are ones that have simulate false information while true assets are ones that dissimulate true information. These true and false assets are placed in a system as directed to by a DT. True assets either remain the same or are hidden by a DT. False assets on the other hand are created by the DT and are not from the original system.

Some DTs consist of the fake assets themselves, describe how to hide assets, or some combination of the 2. Needless to say, they change how the system they are integrated with is perceived, that is their whole purpose. If we consider the art of deception, DTs are assets hidden and shown to create the illusion (this is expanded upon in RQ2).

5.1.3 Style Timeline

Step 1: With a better understanding of how to use DTs, we can begin describing a software system that integrates them. A core principle of deception is that there is what is true, and there is what is false. I make the assumption that a system may be represented as its true self, that is

without deception, and as a false self, that is with deception. When a DT is applied to a system, it takes the true system and transforms it to a false system.

Step 2: I have observed that DTs automatically simulate and dissimulate the assets of a system, where an asset is an arbitrarily granular characteristic (could be a port, space in memory, port, files, features, etc.). Provided that a true system is comprised solely of normal assets, a false system comprises of hidden assets, normal assets, and fake assets.

Step 3: Refining the definition of what a system, it is not composed of assets, it is the collection of assets. Therefore, a true system is the collection of normal assets while the fake system is the collection of hidden, normal, and fake assets.

Step 4: Now the relation between the assets of the true system are mapped to the fake system. DTs automate this process, but hidden assets are created by hiding normal ones and fake assets are added (created).

Step 5: Given that there are both a true system and a deceptive system, we can phrase how a given system is as a reality. One reality gives way to another provided some transition take place, that is, that assets of the system are altered (normally through DT).

Step 6: I added the ability to transition back to previous realities. If assets can be hidden, they can become unhidden. Similarly, if assets are added, they can be removed.

Step 7: Some of the definitions are getting mixed up. A distortion is described as both a process and an effect. Instead, distortion is a process that creates an illusion (the effect). Additionally, instead of using realities which I made on my own, it is a transition from a system without an illusion to one with an illusion.

Post Thesis Defense: The thesis committee did not request any direct adjustments to the style. Rather they requested I update the thesis document itself. The changes consist of more thorough expansions on the evaluation process and the future works section (*Section 6.2*).

5.2 RQ1

RQ1: What concerns should developers have when integrating deception technologies into any software system?

5.2.1 Deployment and Management

It has been noted that the deployment and management of DTs into software systems can be a difficult task [5, 6]. DTs must be maintained just like other software systems, and if done improperly it may tip malicious users or malware that a DT exists. If that DT is discovered, not only is it less useful, but it may even introduce new surface area for hackers to work with.

Through the methodology DTs are described as tools that cause distortions, the result of which are illusions. Breaking down how DTs cause these distortions, it can be seen that they simulate fake components or dissimulate real components. Therefore, the deployment and management of DTs can be described as the deployment and management of components. There is, however, the additional element that is the art of deception.

The reason why DTs are difficult to deploy and manage is not because they are software components, rather it is because they employ the art of deception. DTs must be carefully deployed and managed in software systems to maintain their purpose, that is to change the perception of a given system. This is technically done by deploying and managing components, but the result is the deployment and management of illusions within a software system.

As a very basic example, many software products create releases when changes are deployed to production, perhaps with every deployment to a production environment they increment the version number. If a DT is integrated into a product, deployed to production, and the version number increases, it may partially break the illusion; people will look for what is new because it is known that something was deployed to production. In this situation, a developer should consider not incrementing the version number (at least keeping it private) to better support the illusion created by the DT.

The proposed style does not address the way components are deployed or managed, that is left to the future architectural pattern(s) to describe. What it does address is the idea that components are simulated/dissimulated to create illusions. Illusions are introduced to and removed from systems through the transition called distortion, which is the deployment and management of components.

5.2.2 Art of Deception

To gain security benefits from DTs it is necessary to undergo a process described by the art of deception. Unlike other security mechanisms, it is not always a simple choice of adding a tool

(such as choosing https over plain http). The security mechanism is gained by creating an illusion that is believable and yields some sort of beneficial effect. If the illusion is not believable the desired effect will not take place. On the other hand, if the illusion is believable but does not have a desirable effect then it does not benefit security. As such, there is a whole process to make illusions both believable and worth making.

This process is considered by a few works [1, 2, 3, 5]. The art of deception must be accommodated by the software lifecycle if DTs are being integrated. All of these works understand that a DT has a goal, i.e., a desired effect, that it works to achieve. The goal, when achieved, results in a security benefit that is the effect. To reach such a point, however, it is necessary to plan and use deception carefully, which involves understanding the art of deception.

A great example of a developer who more fully understands this concept is user 0x4D31 on GitHub. He has developed a number of DTs including honeyLambda and honeybits. He created honeybits to allow developers to make higher fidelity illusions. honeybits, as described before, spreads breadcrumbs and honeytokens around a system. honeybits does next to nothing on its own as it only creates faux components that may eat up some time from a malicious user. When used with a honeypot, however, it makes the integrated honeypot more believable by simulating false interactions between the honeypot and the target system.

There is probably no greater place that the art of deception can be applied than software. It is riddled with information and flexibility. There are a few things developers who wish to integrate DTs into their system need to reconcile beforehand. If they wish to integrate DTs as security mechanisms, they need to have some sort of process in their software lifecycle that allows them to design, implement, and deploy illusions into their software systems.

5.2.3 Efficient and Effective Security

It is unknown how effective DTs are at bolstering security in software systems. This is due for a number of reasons some of which being that security metrics can be difficult to work with [8] and that DTs are primarily used for research over production pursuits. Many sources claim that DTs can bolster security of software systems and very few if any state that they should be avoided.

This is a primary reason any developer would integrate a DT into their software system: to bolster security. While some of these DTs may be integrated with ease and little concern for the actual illusion that is taking place, I do believe that it is necessary for developers to better understand the art of deception (if they chose to integrate DTs).

5.3 RQ2

RQ2: How does the architectural style accommodate the different kinds of deception technologies?

The art of deception requires the hiding and showing of information. The information, or assets, that are hidden and shown can be arbitrary. The DTs I have read about and examined can be as small as a password to as large as another software system. While DTs on their own can be very complicated, it suffices for a system to view a DT as a set of hidden and/or fake assets; the target system may run independently from the DTs.

As described in *Section 4.2.3* there are several methods for deploying DTs. Very rarely does a DT require to be built into the foundation of a software system [5]. This supports the idea that a system may exist and operate on its own before integrating DTs.

The specifics of how each DT is integrated can be described based off the simulation/dissimulation of assets. DTs either alter the assets within a given system, that is, a system without illusions, or it introduces new assets to simulate new false information. It is this process of integrating DTs that should be called a distortion. The process of distortion allows the introduction and removal of illusions.

5.4 RQ3

RQ3: What documents and diagrams are necessary to accurately depict the architectural styles?

After consulting with Dr. Mehdi Mirakhorli from RIT I decided to go with the “Gang of Four” documentation approach. The approach allows for an abstract diagram describing how the collaborators relate with one another.

While documenting the proposed style I found it difficult to keep my definitions straight. This was evident when examining the documentation. After several reiterations and recasting’s, it is possible to know whether or not the definitions make sense through this style.

5.5 Architectural Style

Pattern Name: Distortions

Intent: Integrate deception into software systems to create illusions. Allow for the transition between different versions of the system through distortions.

Also Known As: N/A

Motivation: Deception may be integrated into software systems to complement existing security solutions. The practice of deception involves hiding what is true and showing what is false. This practice may be applied to software systems but it requires the developers to understand the nature of deception.

Deception compliments existing security practices and requires careful planning, deployment, and management to be effective. Deception technologies are a great tool for apply deception to a software system, and this style describes how software systems integrate deception technologies for better security mechanisms.

Applicability:

This style may be used when a system integrates deception (often in the form of deception technologies). Helps with the deployment and management of illusions within a software system by describing how distortions result in illusions.

Structure:

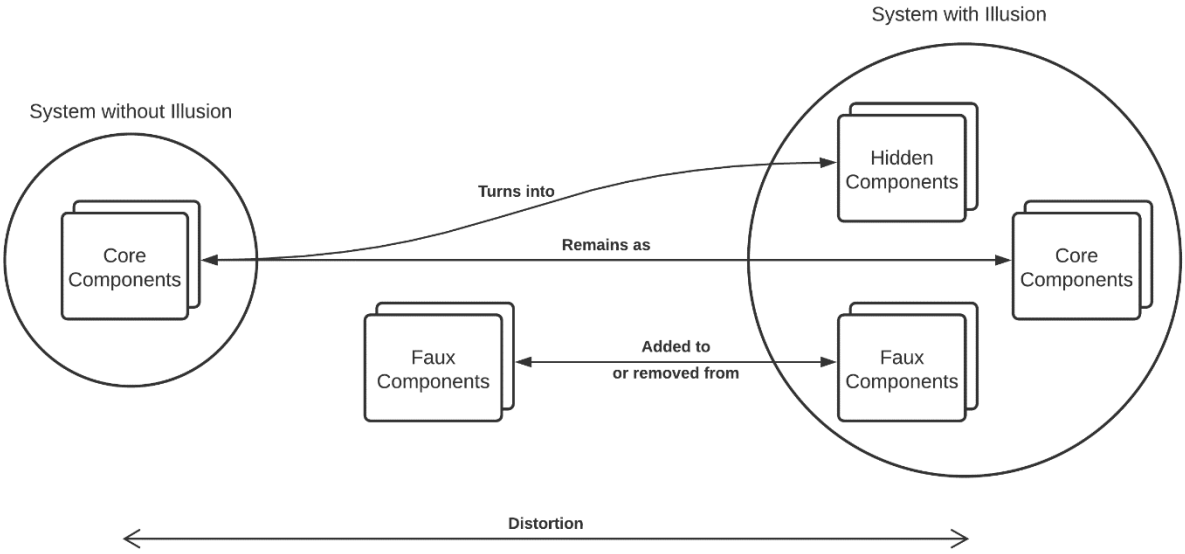


Figure 1

Participants:

- *System without Illusion:* A software structure containing exclusively core components. Deception does not exist within this structure.
- *System with Illusion:* A software structure containing some combination of core, hidden, and faux components. Is a structure that bares at least one illusion.
- *Illusion:* An effect that serves as a security mechanism in the system. Contributes to how a user interacts with the system.
- *Distortion:* The process of deploying and managing components in a system. A distortion is the transition from one system to another.
- *Core Component:* An arbitrarily granular characteristic of the environment. Make up the software structure that is to be protected.
- *Hidden Component:* A core component that has been dissimulated (hidden). It still contributes to the software structure that is to be protected.
- *Faux Component:* An arbitrarily granular false characteristic being simulated (shown). Must not be presented directly to the users of the system.

Collaborations:

A system undergoes a transition to another system through a distortion. The distortion involves deploying and managing the [core, hidden, faux] components of the system. Core components may remain as is or be turned into hidden components. Both core and hidden components directly contribute to the software structure that is to be protected. Faux components may be added or removed from the system to help create illusions. Faux components must never be directly presented to the users of the system.

Distortions are responsible for the illusions that exist in a system. The purpose of the illusion is to yield some sort of security benefit. Should a transition between

Consequences:

- *Increased complexity:* This style will almost always increase the complexity of the system it is applied to. In order to have multiple versions of the system, it is necessary to have some combination of hidden and or fake assets. Depending on the number of possible systems and the intensity of the distortions the structure becomes drastically harder to maintain. This may lead to mistakes when developers work to maintain the system (such as fingerprinting).
- *Adaptable systems:* By transitioning between illusions, it is possible to change the posture of the system. Given that each version of the system serves to create some kind of illusion, it is possible to choose what illusion, if any, will be present in the system. This allows developers to more proactively manage their deceptions in their systems.
- *Emphasis on clear and concise deception:* Illusions describe the set of assets necessary to create different ruses and illusions. Rather than just applying deception to a system in the hopes of security, each illusion specifies the ruse and the intent.
- *Levels of fidelity:* Each asset has a granular level of specificity. Ideally deceptions are made by managing as few assets in the system for efficiency. If so desired, however, developers have the option to pour more resources into different illusions to make them more effective.

Implementation:

1. *Creating alternate illusions:* Each version of the system must have a way of organizing its assets. While it is not required that every system has hidden or fake assets, it must allow for them. The way systems are designed and organized should be specified along with their rational for existing.
2. *How to cause distortion:* After illusions are created there must be a way to transition from one system to another. The way distortions take place should be specified.
3. *When to cause distortion:* When distortions are able to take place the trigger for causing such a distortion must be defined. When, if ever, does a distortion take place and for what reason?

Sample Code and Usage: N/A

Known Uses: N/A

Related Patterns: N/A

5.6 Evaluation

Referring back to my definition of an architectural style in *Section 2*, an architectural style is a conceptual way of viewing, organizing, and developing a system. To evaluate whether or not the proposed style can be considered an architectural style (that is a way of viewing, organizing, and developing a system) I walkthrough 4 different scenarios. In each scenario I describe a software system, provide a DT to be integrated, and explain how the system that integrates the DT adheres to distortions (the name of the proposed style).

The 4 scenarios include: dissimulating a core component, simulating a faux component, faux components replacing core components, then compound simulation and dissimulation. Fundamentally the proposed style involves simulation and dissimulation (showing and hiding information). I have provided a scenario for simulation, dissimulation, and another demonstrating the 2 working together. I included the 3rd scenario where one component **replaces** another to demonstrate why I prevent core components from becoming faux components.

Originally, I had planned to evaluate the proposed style by interviewing a set of cybersecurity experts. This would have provided my thesis with an external evaluation process. However, due to a lack of interviewees and time constraints, I dropped the interview process in favor of this scenario-based evaluation. I discuss this change and its impact on my thesis in *Section 6.12*.

From the following scenarios we can say that the proposed style, distortions, is an architectural style and is applicable to systems that integrate DTs.

5.6.1 Dissimulating a Core Component

This scenario describes how a core component turns into a hidden component.

Let’s say there is a software system where the developers use a logfile for auditing. In the rare case that a malicious user gains access to the files of the target system, the developers wish to maintain the integrity of their logfile. To do so, they decide to dissimulate (hide) the logfile. They take a DT, say a steganography tool, and hide the logfile inside an image on the same system. The logfile is still present and can be retrieved, but is not made easily perceivable to malicious users.

First off, an entire mechanism would need to be designed and put in place to allow the logfile to be hidden and revealed by the developers. This may require the introduction of new core components, i.e., the steganography tool, to allow for the hiding and retrieval of the logfile. If a part of the target system needs access to the information within the logfile, it must also be able to retrieve the information from its hiding place discretely.

This scenario demonstrates that integrating a DT into a software system can be a complicated process. Dissimulating one core component may require adding or changing other core components to accommodate the change. The proposed style does, however, allow for this to take place.

The core component that is turned into a hidden component is the logfile. The logfile, as a hidden component, takes the form of an image. The entirety of the target system is present though it appears differently. Granted that the target system has the ability to hide the logfile in the image and to retrieve it the system behaves as expected. Should a core component need to be added to resolve the functionality the system must expand accordingly.

The logistics of the distortion should be left to the architectural pattern. For the time being, this scenario can be framed in terms of both core and hidden components. Furthermore, the idea that integrating a DT into a software system adds surface area is widely accepted amongst the DT surveys.

5.6.2 Simulating a Faux Component

This scenario describes how to introduce a faux component into a system.

Let’s again say that there is a software system where the developers use a logfile for auditing. They still want to protect it when a malicious user gains access to the target system. This time, they decide to simulate (show) a false logfile. The DT they use is a honeytoken, that when read from or written to, will notify the developers that someone is interacting with the system. This can be broken down into the technical aspects and the art of deception.

The technical aspect of executing this is much simpler than hiding a core component. Instead, the honeytoken, or faux component, only needs to be added to the system and does not interact with any of the core components (though it may act like it does).

This scenario is seamlessly executable through the proposed style. The distortion to the system with an illusion involves adding the honeytoken logfile. The honeytoken logfile is a faux component, it did not exist in the original system as a core component. When and how that honeytoken is introduced to the system is left for the architectural pattern.

5.6.3 Faux Replacing a Core Component

This scenario describes what happens when a faux component replaces a core component.

As an alternative scenario to *5.6.2*, let's say that the developers replace the real logfile with the honeytoken logfile. The distortion from one system to another involves replacing the real logfile for the honeytoken. The core component that was the real logfile turns into the faux component that is the honeytoken.

This scenario does sacrifice the real logfile, the integrity of the system, to make this illusion happen. While there may be specific situations where this scenario is viable, it breaks the constraint of maintaining the CIA of the target system (it is no longer possible to retrieve the real logfile for auditing).

I would say this scenario is a more reckless version of the proposed style, as it allows the developers more freedom in what the distortions cause (at the cost of breaking the CIA of the system). I would therefore consider this outside of scope, or at the most, a variation of the proposed style.

5.6.4 Compound Simulation and Dissimulation

This scenario describes multiple components being simulated/dissimulated.

It is possible to perform the simulations and dissimulations described in *5.6.1* and *5.6.2* in the same system. The hidden logfile (hidden component) and the honeytoken (faux component) can both be in the system at the same time. In this scenario, the developers combine both techniques to contribute to the same illusion. The real logfile is hidden within an image while the honeytoken logfile is presented to the malicious user.

This addresses a flaw in the illusion from both sides. When hiding the logfile, it may raise the question of why there isn't a logfile. On the other hand, presenting an additional decoy logfile in the form of a honeytoken raises the question of why there are multiple logfiles. By using both

simulation and dissimulation, only one logfile is presented to the malicious user. To execute such a scenario, the developer must consider the execution of each part, that is dissimulating a core component and simulating a faux component.

This scenario can take place according to the proposed style because it is framed in terms of core, hidden, and faux components. This scenario demonstrates a system that performs every possible relationship as described by the proposed style. A core component is turned into a hidden component (logfile turns into an image storing the logfile) and a faux component is added to the system (honeytoken logfile). The distortion involves introducing both the hidden and faux components to the system, and the illusion is created through the joint effort of hidden and faux components.

6 Threats to Validity and Future Work

This study has proposed and documented a style for designing a software system that integrates DTs. The proposed style, however, is strictly a theoretical one. This section of the paper will therefore discuss threats to validity and future works for the documented style.

6.1 Threats to Validity

The proposed style is strictly a theoretical one and may or may not exist within the cybersecurity domain. Originally, I had hoped to capture the architectural styles surrounding the integration of DTs into software systems. I refined the scope to proposing a theoretical style due to project constraints and as a way to mitigate some threats to validity. The threats described below raise concerns for how viable and complete the proposed style is.

6.1.1 Research and Analysis

The documented style is abstracted from data collected through open-source repositories and published works. The open-source repositories contain the documentation of DTs while the published works contain developers' concerns and practices. This thesis assumes that the design of a software system, which incorporates DTs, will fall somewhere between the technical requirements of the DT itself, and various documented conventions that developers have. As such the documented style does not reflect the current practices in the cybersecurity domain; rather it represents a possible style created by the technical requirements of DTs and the expected practices of the cybersecurity experts.

The repositories are strictly public and discrete implementations of DTs. This does raise a few concerns for the documented style. The repositories are used to understand the technical requirements of using DTs (i.e., through documentation). Since these are public repositories, they do not represent the entire set of technologies I am interested in. While it is therefore possible that some technical constraints are not being considered because they are hidden as private implementations, this is an acceptable risk. Because of the level of abstraction involved, the differences between open-source implementations of DTs and closed-sourced ones should be near-negligible with regards to style. While I did examine these DTs in these repositories, I did not directly observe them being integrated into software systems.

The analyzed conventions and concerns of developers are based on my literature review and therefore may be outdated or inaccurate. I attempted to mitigate this by using more recent and well cited works.

6.1.2 Evaluation through Scenarios

I had originally planned to interview a set of cybersecurity experts instead of walking through scenarios. Due to a lack of interviewees and time constraints, I dropped the interview process in favor of scenario-based evaluation. This change did impact the process and results of my work. The most immediate consequence was that I was unable to externally validate my style. Additionally, I lost out on the opportunity to validate my findings on the concerns and practices of developers who utilize DTs. I was unable to determine if this style occurs in practice or how cybersecurity practitioners perceive the proposed style.

On the other hand, walking through possible scenarios where the proposed style may be applied gave me the opportunity to provide additional context and examples of the proposed style. This helps better document the style by demonstrating how it may be applied.

6.2 Future Work

Here are some areas where future research may be conducted. These areas are broken down into *Remaining Challenges* and *Research Questions*. *Remaining Challenges* are points of research that can further solidify the findings of this thesis while *Research Questions* are generic points of research related to this domain.

6.2.1 Remaining Challenges

6.2.1.1 Occurrence of the Proposed Style in Practice

The most immediate question that should be addressed is whether or not the proposed style occurs naturally in the software domain. This was decided to be outside of the scope for this study because I had concerns about accessing private repositories and products. I did not have the time or resources to gain access to such information. I therefore suggest that a study should be done on what styles emerge when a software system incorporates DTs. The identified styles could be any existing and documented style (including the one proposed in this paper). I recommend that this be done on both public and private solutions in case of any disparities.

6.2.1.2 *Variations of the Proposed Style*

Variations of this style may also be pursued. As noted during the scenario-based evaluation, I prevent faux components from replacing core components. It may be possible for a variation of the proposed style to allow faux components to replace core components without entirely breaking the CIA of the target system. This is just one possible variation of the style that I theorize. And as such, this possible variation, along with other possible variations, should be identified and analyzed.

6.2.1.3 *Architectural Patterns of the Proposed Style*

Eventually, once this style (or another style regarding the same subject) has been more well established, a study can be done on how it manifests as architectural patterns. This paper discussed at very abstract levels, and as this avenue of research continues it should be moved into a more concrete direction. A good next step after styles would be researching and documenting the architectural patterns that adhere to the proposed style.

6.2.2 Research Questions

6.2.2.1 *In what ways does the proposed architectural style impact the other segments of the software lifecycle?*

After presenting my work to the thesis committee it was brought to my attention that the proposed style only considers the development segment of the software lifecycle. The proposed style, for example, could potentially work very well with the practice of continuous integration and deployment (CI/ CD). This is where software developers and IT staff operate closely together in a back-and-forth relationship. Fully capturing the impact of the proposed style on the other segments of the software lifecycle requires further research.

6.2.2.2 How can we measure the security benefit gained by integrating a deception technology within a software system?

Security metrics on their own can be a complicated matter [7]. Throwing DTs into a software system further complicates the matter. There is the added complexity that the illusion created by the integration of DTs may work against one individual but not on another. The security gained by a DT is highly coupled with the perception of the system by the malicious user. As such, the above research question was identified to help research ways of measuring the effectiveness of DTs that are integrated into software systems.

7 Conclusion

I have examined the art of deception and the creation of DTs for the purpose of integrating deception into software systems. The art of deception should be considered a separate discipline that is used by developers to achieve certain security benefits. This discipline has deep roots in warfare and can be traced back hundreds of years ago. It is still a very useful skill to know, though it requires a certain amount of experience and intuition to use effectively.

The proposed architectural style (“Distortion”) describes how software deployment relates to the art of deception with special interest in security. Software components are both hidden and shown to create illusions, influencing how users interact with the system. There are many aspects of this compound discipline, the joint effort of software and the art of deception, that needs researched and documented.

Annotated Bibliography

Cheating and Deception
A book on deception as a discipline. Its focus is on the nature of deception: who, what, when, where, and why deception. While this work does not recognize software or deception technologies, many published works in the deception-cybersecurity domain accept and build off the principles in this book. I also accept and use parts of their view on this discipline.

[1] J. B. Bell and B. Whaley. Cheating and Deception. Transaction Publishers New Brunswick, 1991.

Honeypots: Tracking Hackers
One of the first published works on deception in software, this book describes the nature of honeypots. Honeypots were prevalent before this book was published, but Lance Spitzner sorted out and documented many of the practices at that time. Also, since this is one of the first books on deception technologies, he carefully considers what deception is. I touch on a number of

[2] Spitzner, L. (2003). Honeypots: Tracking hackers. Boston: Addison-Wesley.

Planning and Integrating Deception into Computer Security Defenses
Outlines a general process for integrating deception technologies into software systems. The process was designed incorporating deception principles along with general software development practices. While a good step in the right direction, I found that it leaned more heavily on the practice of deception and not enough on the logistics for integration.

[3] Mohammed H. Almeshekah and Eugene H. Spafford. 2014. Planning and Integrating Deception into Computer Security Defenses. In Proceedings of the 2014 New Security Paradigms Workshop (NSPW '14). Association for Computing Machinery, New York, NY, USA, 127–138. DOI:https://doi.org/10.1145/2683467.2683482

Demystifying Deception Technology: A Survey
A quick overview of the types of deception technologies and how they are used in security. After a general discussion of the practice of deception technologies, they specify and explore the honey prefix categories (i.e., honeypots, honeynets, honeytokens, etc.).

[4] Fraunholz, D., Anton, S. D., Lipps, C., Reti, D., Krohmer, D., Pohl, F., . . . Schotten, H. D. (2018). Demystifying Deception Technology: A Survey. doi:1804.06196

Deceptive Techniques in Computer Security: A Research Perspective
Examines types of deceptions and how they manifest as concrete deception technologies. The researchers do a great job of considering the technical aspects for using these technologies in software systems. In the process of examining deception technologies, they provide a number of ways to categorize the technologies. In addition, they discuss concerns and considerations for when developers use the technologies.

[5] Xiao Han, Nizar Kheir, and Davide Balzarotti. 2018. Deception Techniques in Computer Security: A Research Perspective. ACM Comput. Surv. 51, 4, Article 80 (September 2018), 36 pages. DOI:https://doi.org/10.1145/3214305

A Survey on Honeypot Software and Data Analysis
After examining a large dataset of honeypots and categorizing the technologies, the researchers discuss how vastly different one honeypot can be from another. Through this research they provide their taxonomies and various categories of honeypots.

[6] Nawrocki, M., Wahlisch, M., Schmidt, T. C., Keil, C., & Schonfelder, J. (2016). A Survey on Honeypot Software and Data Analysis.

A Survey on Systems Security Metrics
While not on deception technologies, this paper discusses the difficulty of fully capturing the quality of security for a system through metrics. This contributes to the difficulty of accurately capturing the benefit of deception technologies as security mechanisms in software systems.

[7] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. 2016. A Survey on Systems Security Metrics. ACM Comput. Surv. 49, 4, Article 62 (February 2017), 35 pages. DOI:https://doi.org/10.1145/3005714

Strategic Defense and Attack in Deception based Network Security
A game theory proposal on deception technologies as security mechanisms. Fraunholz and Schotten describe the interaction between the attacker and the defender and how they take various aspects of the attack into account. This is primarily used as an example of how deception can quickly become complicated in a system.

[8] D. Fraunholz and H. D. Schotten, "Strategic defense and attack in deception based network security," 2018 International Conference on Information Networking (ICOIN), Chiang Mai, Thailand, 2018, pp. 156-161, doi: 10.1109/ICOIN.2018.8343103.

Uncovering the Honeypot Effect: How Audiences Engage with Public Interactive Systems
There is a connection between the technical feat of applying deception in software and the social sciences of the nature of deception. This is a social experiment rather than a software study whose purpose is to research how to get various passersby to interact with honeypots (in this paper a honeypot is an activity).

[9] Niels Wouters, John Downs, Mitchell Harrop, Travis Cox, Eduardo Oliveira, Sarah Webber, Frank Vetere, and Andrew Vande Moere. 2016. Uncovering the Honeypot Effect: How Audiences Engage with Public Interactive Systems. In Proceedings of the 2016 ACM Conference on Designing Interactive Systems (DIS '16). Association for Computing Machinery, New York, NY, USA, 5–16. DOI:https://doi.org/10.1145/2901790.2901796

Ninth Annual Cost of Cybercrime Study
I've researched the issue of cybercrime and use a number of statistics from this source. This is to help ground my issue in clear and concise terms.

[10] Bissell Senior Global Managing Director – Accenture Security, K., LaSalle Managing Director – Accenture Security North America, R. M., & Dal Cin Managing Director, P. (2019, March 06). 2019 Cost of Cybercrime Study: 9th Annual. Retrieved September 20, 2020, from https://www.accenture.com/us-en/insights/security/cost-cybercrime-study

Gang of Four Template
I pulled the "Gang of Four" design template from the Hillside Group website to document the proposed style.

[11] Gang of Four template. (n.d.). Retrieved March 15, 2021, from https://hillside.net/index.php/gang-of-four-template

Victory and deceit: Deception and trickery at war
Used by Fraunholz et al. in "Demystifying Deception" to categorize the types of deception technologies.

[12] James F. Dunnigan and Albert A. Nofi. Victory and deceit: Deception and trickery at war. Writers Club Press, San Jose [etc.], 2nd ed. edition, op. 2001.

Glossary

Architectural Pattern: A concrete way to implement an architectural style. Solves issues raised by the overarching architectural styles (ex: proxy).

Architectural Style: A conceptual way of viewing, organizing, and developing a system (ex: client-server).

Art of Deception (AoT): the discipline of distorting reality, of creating illusion, and getting people to believe that what they perceive is real.

Component: an arbitrarily granular piece or characteristic of a software system (could be a binary digit or an entire subsystem).

Deception Technology (DT): a software tool that simulates/dissimulates components as a security measure in software systems.

Dissimulation: The act of hiding a core component, which contributes to an illusion.

Distortion: a transition from one perceivable system to another (through simulation/dissimulation).

Illusion: a set of simulations/dissimulations that create some sort of effect(s) when accepted by an individual.

Simulation: The act of showing a faux component, which contributes to an illusion.

Appendix A

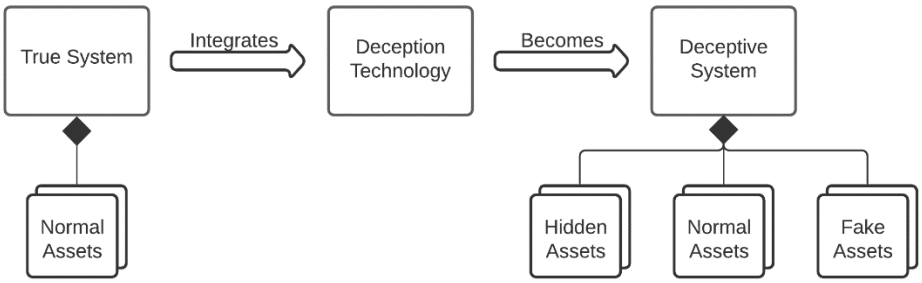
Name	Type	Simulation/ Disimulation	Characteristics	Ruse	Goal	Layer	Deployment	Last Updated
sslab-gatech/aslr-guard	ASLR	Masking	Address space of code	Unappealing	Prevention	System	Added-to	24-Feb-16
kengiter/RuntimeASLR	ASLR	Masking	Address space of code	Unappealing	Prevention	System	Added-to	28-May-16
jaebaek/SGX-Shield	ASLR	Masking	Address space of code	Unappealing	Prevention	System	Added-to	15-Apr-18
immunant/selfrando	ASLR	Masking	Address space of code	Unappealing	Prevention	System	Added-to	16-Dec-19
RobinDavid/LSB-Steganography	Steganography	Repackaging	Images	Unnoticed	Mitigation	Data	Added-to	12-Jun-20
livz/cloacked-pixel	Steganography	Repackaging	Images (encrypted)	Unnoticed	Prevention	Data	Added-to	1-Jun-17
ragibson/Steganography	Steganography	Repackaging	Images	Unnoticed	Mitigation	Data	Added-to	4-Jan-21
lukechampine/jsteg	Steganography	Repackaging	Images	Unnoticed	Mitigation	Data	Added-to	1-Jun-19
syvaidya/openstego	Steganography	Repackaging	Images (hidden watermark signature)	Unnoticed	Detection	Data	Added-to	16-Jan-21
girishsg24/Moving-Target-Defense-RHM-using-SDN	MTD	Masking	Software Defined Network (SDN). IP addresses	Unappealing	Prevention	Network	In-front of	18-Mar-18
Pwdrkeg/honeyport	Honeyports	Decoying	Ports	Benign	Prevention	Network	Isolated	22-Oct-15
securitygeneration/Hon	Honeyports	Decoying	Ports	Benign	Prevention	Network	Isolated	22-Feb-17
mushorg/conpot	Honeypots	Mimicking	Characteristics of a debian system	Benign	Detection	Data	Isolated	11-Mar-21
desaster/kippo	Honeypots	Mimicking	Characteristics of a debian system	Benign	Detection	Data	Isolated	30-Sep-16
cowrie/cowrie	Honeypots	Mimicking	Characteristics of a debian system	Benign	Detection	Data	Isolated	19-Mar-21
honeytrap/honeytrap*	Honeypots	N/A	Framework for creating and configuring many honeypots.	N/A	N/A	Network	Isolated	10-Dec-20
Plazmaz/MongoDB-HoneyProxy	Honeypots	Mimicking	MongoDB database	Benign	N/A	Data	Isolated	5-Apr-20
secureworks/dcept	Honeytokens	Inventing	Credentials stored in memory	Desireable	Detection	System	Isolated	22-Sep-16
0x4D31/honeyLambda	Honeytokens	Decoying	AWS HTTP endpoints	Desireable	Detection	Data	Added-to	19-Oct-18
0x4D31/honeybits	Honeytokens & Breadcrumbs	Inventing	bash_history commands, files, ARP table entries, browser history, registry keys	Desireable	N/A	Data	Added-to	19-Mar-19

Appendix B

A True System is a system without any deception. When a Deception Technology is integrated with a True System it becomes a Deceptive System.

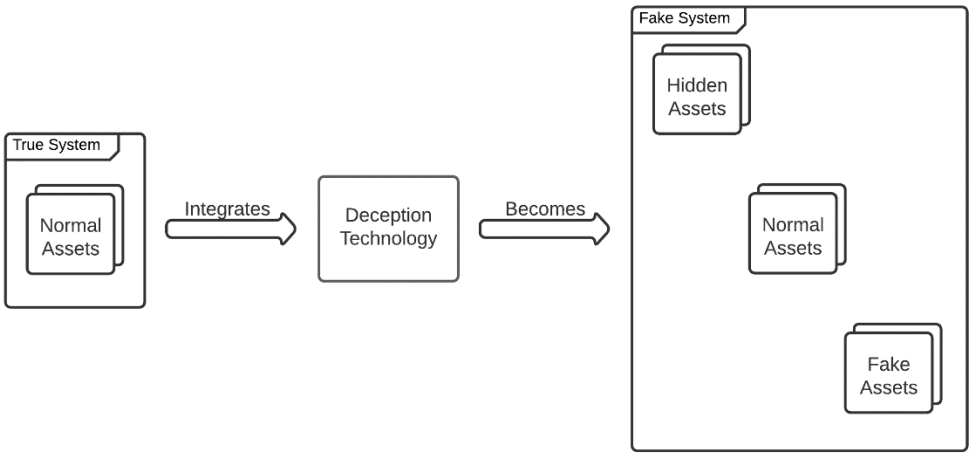
Step 1

A Deceptive System is a True System that has been simulated/disimulated in some way. This is down by hiding what is real and/or showing what is fake.



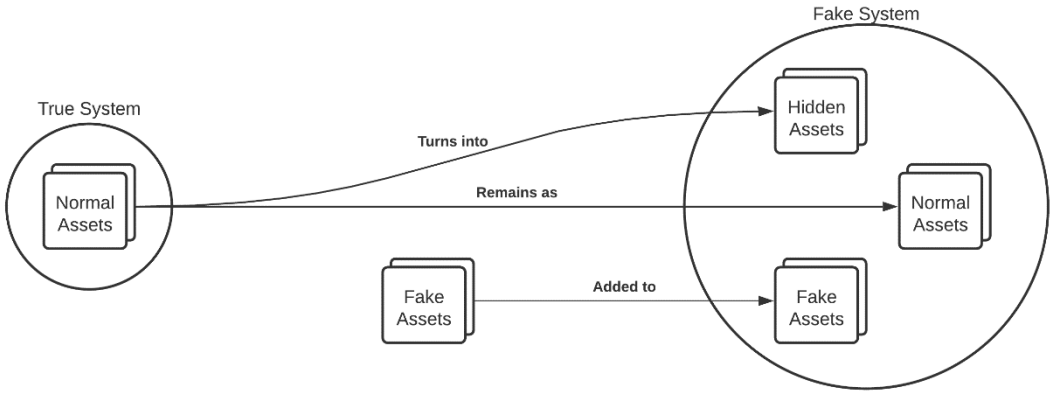
Step 2

Instead of saying a System is composed of assets, express the system as a collection of assets.



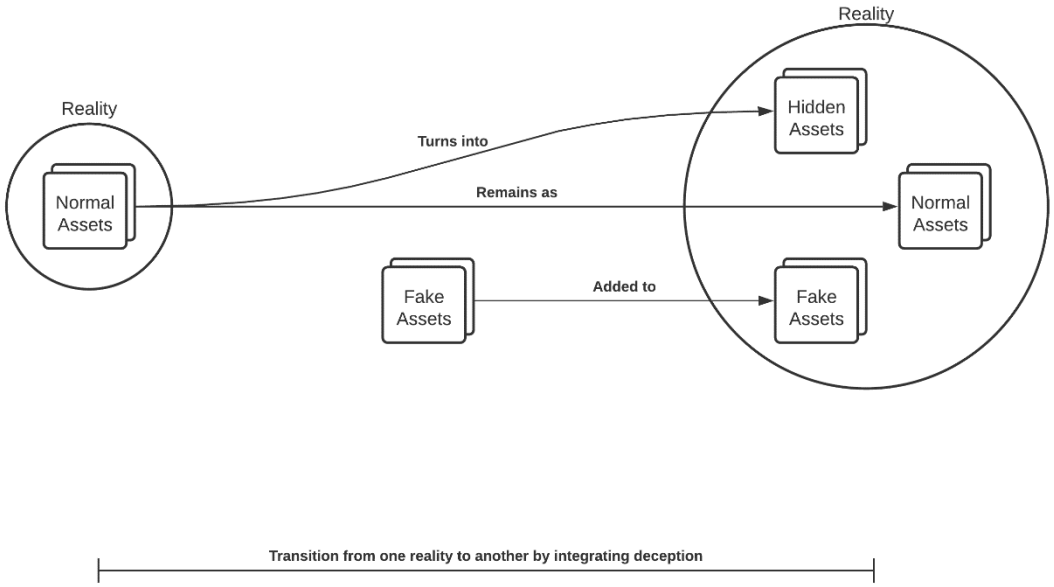
Step 3

Describe what a DT does to an asset to make it hidden or fake.



Step 4

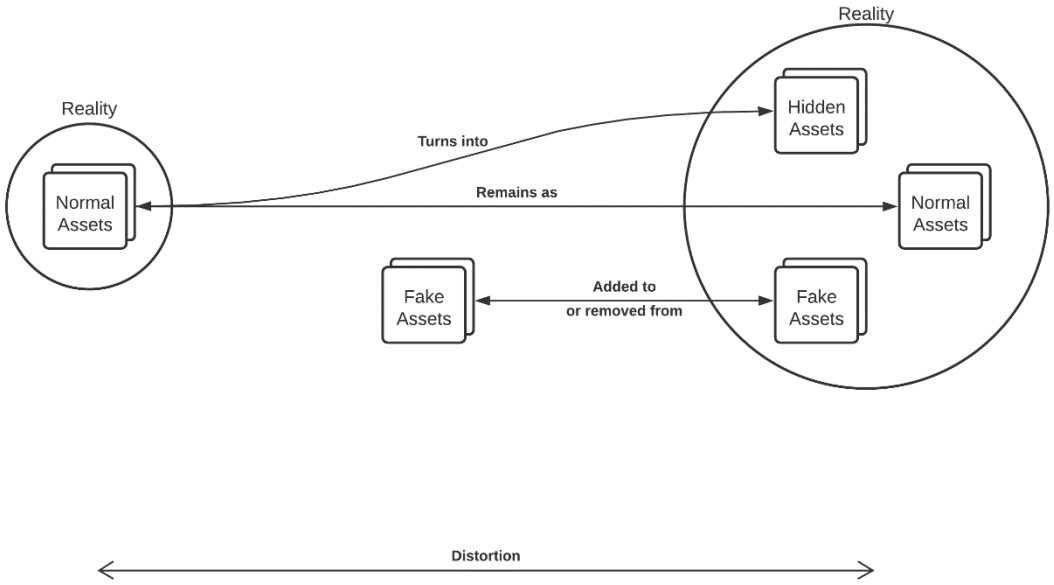
The circles are the system. The arrows are a transitional declaration between different states/versions of the system.



Step 5

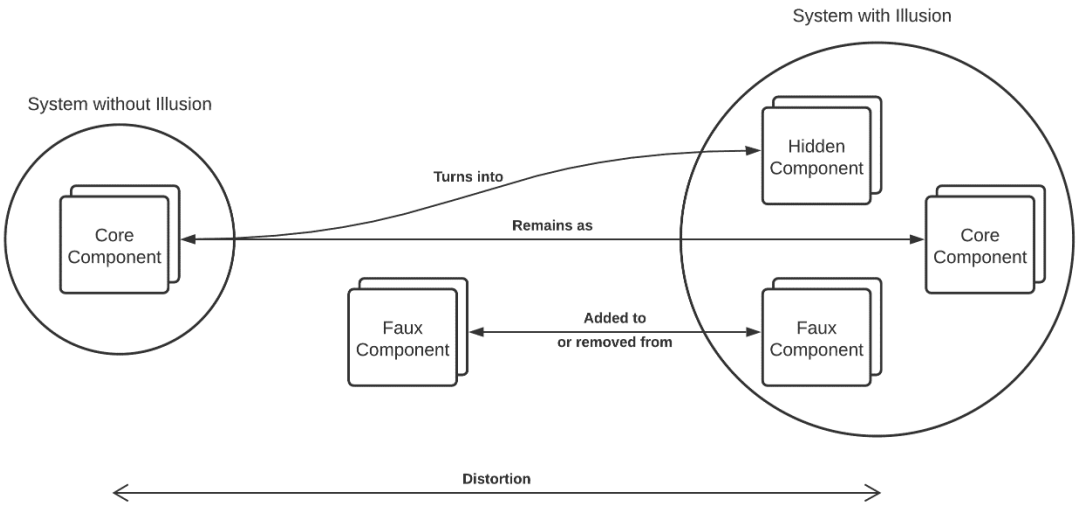
Allow a return to the previous reality should the deception prove ineffective.

This was the first proposed style.



Step 6

After consulting with Kal Rabb, some of the collaborators don't make the most sense. What is a reality and where did the name come from? I update some of the collaborators here.



Step 7