Paths Not Taken: a Secure Computing Tutorial

William Earl Boebert
Retired[1]

Abstract

This paper is a tutorial on the proven but currently under-appreciated security mechanisms associated with "tagged" or "descriptor" architectures. The tutorial shows how the principles behind such architectures can be applied to mitigate or eliminate vulnerabilities.

The tutorial incorporates systems engineering practices by presenting the mechanisms in an informal model of an integrated artifact in its operational environment. The artifact is a special-purpose hardware/software system called a *Guard* which robustly hosts defensive software.

It is hoped that this tutorial may encourage teachers to include significant past work in their curricula and students who are self-teaching to add that work to their exploration of secure computing.

Organization of the Tutorial

This tutorial is divided by topic area to facilitate incorporation in curricula and to assist those who are using it to self-teach. The topic areas are arranged in order of detail, from the most general to the most specific. The first area, Concept of Operations, discusses the issues of systems engineering and the importance of understanding and exploiting the environment in which systems operate. The second section, Models, discusses mental models and the use of abstraction in understanding computer systems. An overview of the Guard model is then given. The Mechanisms topic area is the most specific, where the known technology is described both in detail and in the context of the systems design given in the earlier sections. The final section offers suggestionss to those who may be interested in carrying this work further into an operational system.

Concept of Operations

No system can solve every aspect of a given problem, and experience has shown that explicit prior consideration of focus, limits, and approaches will often save time, money, and blood. A major goal of the tutorial is to show how an understanding of the context in which a system operates influences the design of mechanisms. To do so requires presenting a hypothetical operating environment for the hypothetical Guards.

A proven systems engineering [1] technique is to define an operating environment and a system's place in it by producing a Concept of Operations, or Conops [2]. Three Conops-level topics are included to help the student understand the rationale behind the mechanisms: a statement of problem the Guard is intended to solve, the general approach taken for the solution, and the goals of the assurance process.

Statement of the Problem

A generic network of interest to attackers is one whose purpose is to provide economic or social value. It does this by running applications on general-purpose platforms which directly or indirectly access the internet. The platforms are typically feature-rich operating systems such as Windows, Linux, or Mac OS. The applications may be a mix of locally developed, pur-

chased, and open source software. Some networks, such as those which control processes such as water supplies and electrical distribution, will be comprised of specialized platforms and applications.

With rare exception, such networks are designed and administered with the applications as a primary concern and defense against attack as secondary. This setting of priorities encourages attackers who already enjoy inherent advantages: attackers can choose the time, place, and nature of an attack and they only need to find one exploitable fault, while defenders must cope with all known and lurking ones, and attacker's motivation is enhanced by the often overlooked fact that for many individuals attack is an adventure while defense is just a job.

In today's world attackers may also be directly or indirectly protected from retaliation by sovereign states. They will, in general, be adequately funded either by sponsors or by the unregulated transnational money flows made possible by cryptocurrency [3]. Being funded enables them to be persistent, to perform repetitive attacks and learn from each one. And attackers can exploit features of the internet which permit anonymous action at a distance.[4].

General Approach

The sole purpose of a Guard is to host Security Services code. Development of that code is hosted elsewhere. There is also no requirement to support a browser, or data bases, or any other form of productive application.

The isolation and concentration of Security Services in the Guard nodes means that Guards will be the primary targets of competent attacks, for their defeat would leave the network largely undefended. The design, implementation, and administration of Guard nodes recognizes this fact. Design and implementation places resistance to attack as a requirement above all others, and administration recognizes that social engineering and supply chain attacks will accompany direct attacks on the network.

Separation of Security Services from applications is motivated by the radically different characteristics of the two classes when viewed from a life-cycle perspective. Application platforms are large, complex bodies of software which are subject to structural decay [5] unless updated carefully, which means at wide intervals. When new attacks appear, however, Services code must be updated with all deliberate speed so that attackers have the smallest practical window of exploitation. Putting Services code on the same platform leads to one of two undesirable options: if the Services+applications platform is updated at the rate required by Services, the application is at significant risk of decay. If the Services+application platform is updated at the careful pace necessary to prevent decay, updated Services code will be delayed and its effectiveness diluted.

Figure 1 depicts potential applications of Guard instances to network defense.
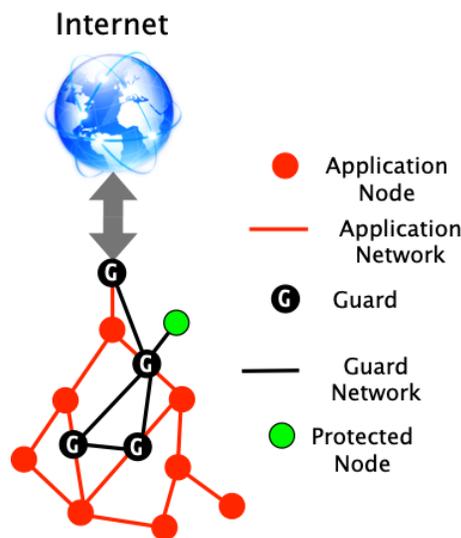

Figure 1: A Network With Guards

The most basic application is a traditional firewall that filters packets and raises alarms at the juncture with the internet. Similar tasks could be carried out within the application network to detect or prevent adverse interactions between application nodes. Guards could interact with each other by using out-of-band communication or virtual private network

(VPN) technology. A Guard could protect a critical but vulnerable internal node such as a machine learning system for intrusion detection, a network attached storage for backups, or a "honeypot" to trap attackers. In all cases the relevant Security Services are determined by the combination of applications and known attacks and therefore system-specific.

The isolation of the Guards from vulnerable network elements is an adaptation of a design principle called *Zero Trust*:

The Zero Trust Model is simple: cybersecurity professionals must stop trusting packets as if they were people. Instead, they must eliminate the idea of a trusted network (usually the internal network) and an untrusted network (external networks). In Zero Trust, all network traffic is untrusted. [7]

Assurance

Assurance determines how human administrators view the cyber systems they administer. Assurance is essential to effective response. Administrators who are assured of known system behavior are more likely to respond decisively and correctly to alarms and other indicators of potential attacks.

There are two assurance goals associated with a Guard. The first is that the internal mechanisms of the Guard operate as advertised even when attacked. The second goal is to export assurance to the Services code it hosts and from that to the administrators. The second, external assurance goal can be somewhat flippantly expressed as "WYSIWYG^2." In more dignified terms, it is the assurance that the semantics of the source code for Services will be enforced by every single instruction executed by the Guard. That assurance enables individuals who are reading the source code for the Services to do so with confidence.

These assurance goals determine the nature of the mechanisms in the model and its structuring. More primitive elements are subject to the greatest amount of testing and analysis. These results then form a basis for arguing that the higher level elements are cor-

rect, in the way that lemmas contribute to theorems in mathematical reasoning.

It should be noted that the assurance exported by the Guard to the Services does not guarantee or even imply that a particular Service is effective. That final, third form of assurance must be provided by those who write and administer the Service.

It should also be noted that the Guard itself is security policy agnostic. Any rules or restrictions to be enforced on application data are the responsibility of the application-specific security services hosted on the Guard.

Models

Models of Nature

The role of modeling in science was described by Arturo Rosenblueth and Norbert Wiener in the 1940s:

No substantial part of the universe is so simple that it can be grasped and controlled without abstraction. Abstraction consists in replacing the part of the universe under consideration by a model of similar but simpler structure.[8]

Similar observations about models were made and expanded by Alan Turing in the preface to his last scholarly paper :

This model will be a simplification and an idealization, and consequently a falsification. It is to be hoped that the features retained for discussion are those of greatest importance in the present state of knowledge.[9]

Note the depth of insight revealed by Turing's phrase "it is to be hoped." All models are incomplete, and the nature and degree of what is left out is arbitrary.

Models of Computer Systems

Turing, Wiener, and Rosenblueth wrote these words before the advent of programmable computers, programmers, and software. That technology requires its practitioners to operate in an arena largely free from physical constraints, as described in a classic passage by Fredrick Brooks:

The programmer, like the poet, works only slightly removed from pure thought-stuff. [...] Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.[10]

Brooks goes on to discuss how these characteristics of software make it so difficult to get right. Over the years, practitioners coped with that difficulty by adopting, mostly without explicit thought, an approach that reversed the sequence of scientific modeling: instead of describing an existing tangible object in intangible terms, they create abstract models *in advance* as a way of discussing a largely intangible software/hardware artifact that does not yet exist, and reason about what they are doing in terms of those models. A particular system may be modeled at progressively greater degrees of detail until the point at which a string of ones and zeros is loaded into silicon and activity is generated. In informal terms it is "models all the way down," and the choice given to practitioners is not whether to use a model but rather what kind of model to use.

The model presented in this paper is, as described above, necessarily incomplete. The "features retained for discussion" are those which give a Guard the ability to resist attack. These are the mechanisms of *structured memory, demand linking,* and *layer enforcement,* which will be described later. Any hypothetical implementation of a Guard would involve filling in the gaps and manifesting the result in software and hardware. It is the intent of the model that such an activity should require no more than understanding the above elements of known technology.

The Guard Model

The Guard has the general form of a resource-management operating system, and is described in terms of layers of functionality [11], as shown in Figure 2.
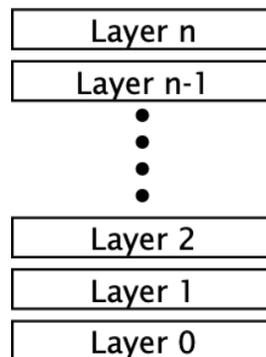

Figure 2: A Layered Structure

A "Layer" in the Guard model is a cluster of data and code objects devoted to a common purpose, such as a file system. When a program in execution (process) has its execution point in procedure belonging to a Layer it is said to be "in" the Layer. An object "belongs" to a Layer if being there permits execute access (for code objects) or read+write access (for data objects.) These concepts will be discussed in detail later.

Layers are used to apply structure to the complex interactions that exist in resource-management operating systems. The structure, which is motivated by a principle called "separation of concerns," [12] is intended to improve understanding and support the structured assurance goal described earlier.

The Guard model carries layering further and provides mechanisms to enforce the Layer structure. These will also be described later.

There a variety of ways in which Layers could interact, such as procedure calls, cooperating sequential processes, or shared data objects. All of these forms of interaction between Layers can be assigned the directional attribute of *dependency.*

Despite it being around for 50 years, there is no industry standard terminology for this relationship. Words like "depends upon," and "uses" capture the essence.[13] Analyses of dependency are employed in Failure Mode and Effects Analysis (FMEA)[14] and Fault Tree Analysis (FTA)[15]. In the latter cases the emphasis is on the important question of what happens if a failure occurs in a particular element.

The term "influences" turns the relation around: B influences A if a difference in B causes a difference in A. This paper will employ whichever term leads to the simplest description.

Mechanisms by Layer

The platform software can then be separated out and refined into three Layers* as shown in Figure 3.
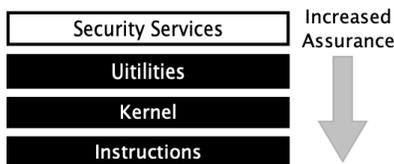

Figure 3: Platform Hierarchy

The model assumes that in any hypothetical implementation the amount of testing and analysis (and the resulting assurance) would increase as one goes down the Layers.

The general assignment of mechanism to Layers is as follows:

Security Services: This is the software that provides the network defense. If one characterizes a Guard as an operating system, this software is the equivalent to the applications that run on it. It is assumed that this is a highly dynamic Layer, with modifications made in response to the unannounced arrival of new attacks.

*If a common term is capitalized in this paper (e.g. Layer) it denotes the Guard definition; lower case denotes the generic meaning.

Utilities: Mechanisms in this Layer are invoked by procedure calls in the security service code. This Layer includes a file system that associates symbolic names with units of structured memory. It also includes the network interface and things like event data recording or logging to preserve and protect event sequences for forensic use. These are well-known functions and accordingly are not treated in detail in the model.

Kernel: This Layer manages the internal data upon which the proper operation of the Instruction Layer depends. Functions in this Layer are invoked by procedure calls in the Utilities Layer.

Instructions: The mechanism in this Layer are invoked by instructions which mimic the interface to conventional hardware with a command, working, and addressing registers. One portion of this Layer provides basic instructions such as arithmetic, test and branch and so forth. The other portion provides security instructions which deal with memory safety and constraints on instruction sequences.

Dependencies

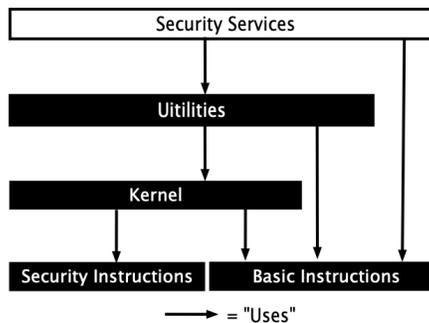The dependencies between Layers are shown in Figure 4.

Figure 4: Dependency Restrictions

This figure shows that the Security Services may use the Utilities and the basic instructions provided by the silicon. The Utilities may use the facilities

of the Kernel and the basic instructions. The Kernel, and only the Kernel, is permitted to use the restricted security instructions. These restrictions on dependency are enforced by the mechanisms incorporated in the model.

The dependency restrictions enforce a design principle which states that no element of the Guard shall depend upon an element of lower assurance. This principle supports the structured assurance effort.

Operational Context

A basic tenet of systems engineering is that the context in which a system operates must be a factor in its design, and documented in the Conops. In the case of the Guard, that context includes the personnel who build it and those who administer it. These are shown in Figure 5.
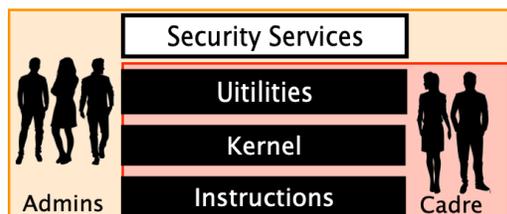

Figure 5: Human and Technical Trust Boundaries

The code that provides the Security Services and the personnel who develop and administer that code form the outer enclave. The code that supports those Services and provides the resistance to attack and the personnel who develop and maintain that code form the inner one. Each enclave grants limited trust to the other.

Service code is reactive in nature and constantly changing, with a premium placed on speed of implementation and updating. To be effective it must be "quick and dirty" and therefore assumed to contain errors and possibly subversions. Lower Layer code is carefully planned and stable with a premium placed on assurance, with secrecy of design as a secondary protection. On the other hand, Service code may

contain information that the organization using the Guard may wish to hold closely.

Outside the administrative enclave all network elements are presumed compromised and hostile, so there is assumed to be a communication channel between administrators and the Guard. Such a channel would be out-of-band, secured by cryptography and employ some convenient device physically controlled by administrators. This channel is not included in the model, but a general outline of functions and mechanisms may be found in [16].

Mechanisms

The primary mechanisms of the Guard model are *Structured Memory, Demand Linking*, and *Layer Enforcement*. These were chosen to show how mechanisms can complement each other in a specific operational environment to provide the emergent property [17] that the "WYSIWYG" assurance goal of the system is met. Each mechanism is a variant of something that has appeared in one or more predecessor system[3], and has been modeled as simply as possible to illustrate the mechanism's principles.

The mechanisms are defined in terms of two system primitives: *Segments* and *Processes*.

Segments

A Segment is an addressable sequence of bytes. All addressing in a Guard is indirect. As shown in Figure 6, places an intermediary element between an instruction's reference to a specific part of memory and the value stored there.

There are several reasons for using indirect addressing, such as flexibility in managing memory, but the one of greatest significance to resistance to attack is the placing of metadata in the intermediary element. "Metadata," in this context, means data *about* data, security-relevant descriptions such as length or whether the data can be interpreted as instructions. Putting the metadata in the address path provides

Direct Addressing

Op Code	Direct Address

Instruction

↓

| Byte |

Indirect Addressing

Op Code	Indirect Address

Instruction

↓

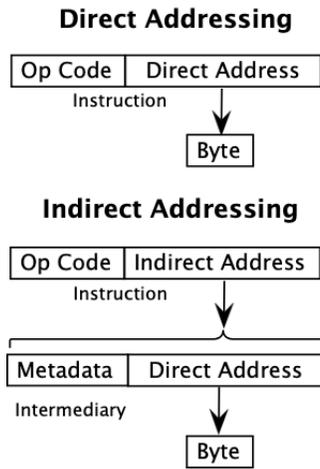| Metadata	Direct Address

Intermediary

↓

| Byte |

Figure 6: Indirect Addressing

high assurance that it will uniformly be encountered by the Instruction Layer.

The intermediary, metadata-holding element in the Guard model are *Segment Descriptors*. Individual bytes are addressed by Descriptor, *Offset* pairs.

Processes

A Process is modeled as an execution point moving through successive object code Segments. Movement from code Segment to code Segment is achieved through the traditional call/return mechanism and pushdown stack.

Processes begin life in the Services Layer. As a Process moves from code Segment to code Segment it may move from Layer to Layer. A Layer Register is used to track the current Layer the Process is in. The Guard model describes a separate pushdown stack for each Layer.

The Guard model explicitly and deliberately does not include Process muliplexing.

Process multiplexing is the technique dividing multiple program sequences into fragments and interleaving the fragments into a single sequence, as shown in Figure 7.
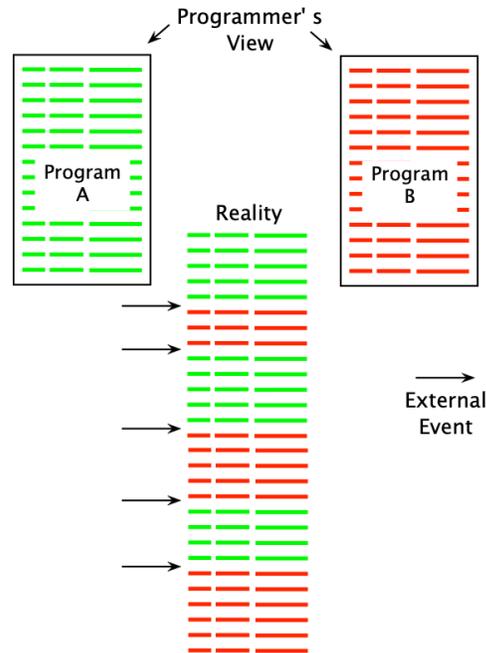
Figure 7: Process Multiplexing

Simulated parallelism contradicts the "WYSIWYG" assurance goal. A programmer or analyst reading a program text will naturally assume that the sequence shown in the text is that which occurs at the Instruction Layer of the Guard. Process multiplexing invalidates that assumption[4].

If parallelism is needed then the design approach is to add entire Guards with a shared memory and synchronize them with messages[11]. This gives each element in the parallel structure the full protection benefit of Layers and "WYSIWYG" assurance.

Processes do not execute on behalf of human users and there are no user-settable access rights in a Guard. Guards are not intended to run productive applications like browsers and data bases. As a consequence Guard Processes would, in general, be simple polling loops or linear code sequences that would sleep until woken by an external event. In traditional operating system terms, all Processes are "daemons" [25].

7

Individual instructions in a Process's sequence have the ability to invoke a special facility called a *Trap*. The Trap mechanism[5] is implemented at the Instruction Layer. It enables a special class of events to be handled at a lower Layer without explicit call/return sequences.

As shown in Figure 8, an Instruction initiating a Trap causes a break in the code sequence, saving of the Process state (register contents and stack) and direct transfer of control to a code segment called a *Trap Handler*. The illustration shows a Trap to the Kernel Layer; Traps to the Utility Layer are also possible.
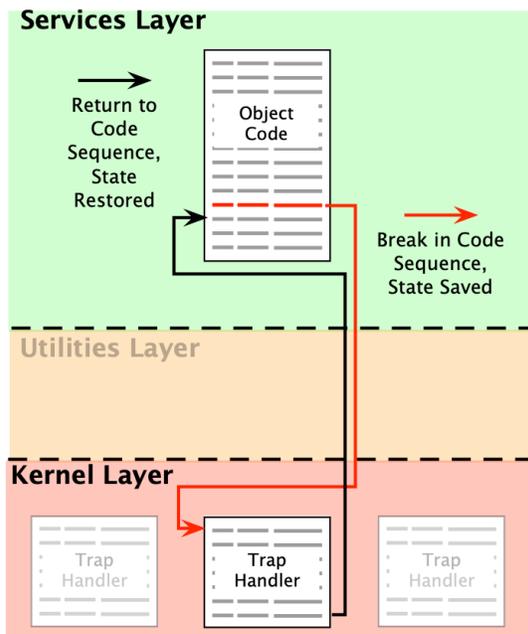

Figure 8: The Trap Mechanism

The kind of Trap initiated determines which Trap Handler is invoked. The Trap Handler performs Trap-specific actions and then restores the Process state and resumes execution at the instruction immediately after the one that initiated the Trap. Trap Handlers thereby become, in effect, software-implemented extensions to instructions[6].

Traps can be initiated explicitly by a "Trap" in-struction, as a response to an illegal or impossible action such as divide by zero, or when the attempt to fetch a byte encounters a special value.

Structured Memory

Structured memory is at the heart of the interaction between processes and segments. It has two aspects: the format and meaning of Descriptors, and the manner in which a Descriptor address is mapped onto the conventional primary and secondary memories at the Instruction Layer[7].
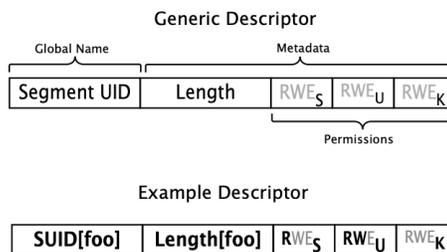
Descriptors

Figure 9: Descriptors

The generic form and an example of a Descriptor is shown in Figure 9. Descriptors have two main fields, a *Segment UID* or *SUID* which uniquely identifies the Segment, and a set of metadata. The inclusion of metadata makes the Guard model an example of a "tagged architecture," [26] where the tags apply to segments rather than words or bytes.

The metadata consists of a length field, whose inclusion and checking by the mapping logic makes memory safety [27] automatic, and a set of permissions. There is one permissions field for each Layer. The values shown (read, write execute) are examples and other values, such as "append" are possible. There is no manual setting of permissions in the model; all permissions are determined at the time the Guard software is initialized prior to installation.

When permission violations are detected by the

mapping logic, it generates a Trap to an error Trap Handler which will perform appropriate actions like sending an alarm to administrators and halting.

The example Descriptor in Figure 9 is for a data Segment called "foo," which will be used as an example throughout the discussion of mechanisms. It is assumed to be managed by a Utilities Layer routine called "foo_owner."

The "SUID[foo]" value denotes its identity and "Length[foo]" its size. It is a Utility Layer Segment, and this is shown in its permissions. The first permission field says that a Process executing in the Services Layer has "read" only access to "foo," one in the Utilities Layer has "read" and "write," and one in the Kernel Layer is denied all access. This would be a typical set of permissions for something like a file system Segment.
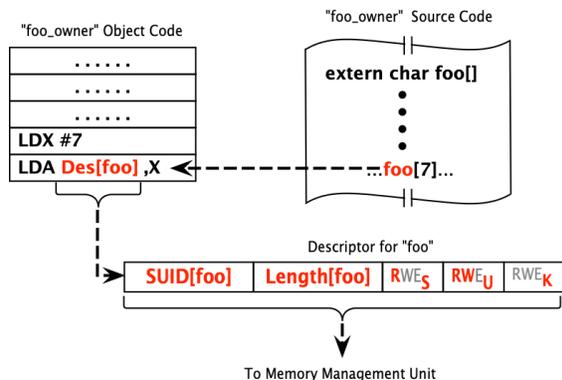
Descriptor Addressing

Figure 10: Ideal Name Association

Figure 10 continues the example by showing the desired relationship between the three levels of names that connect source code to physical memory. The source program for "foo_owner" contains the declaration of "foo" and a reference to a byte at the 8th position in "foo" by means of an indexed instruction:

LDX #7: Load index register with constant "7"

LDA Des[foo],X: Load accumulator with byte at physical address defined by the Descriptor for foo offset by contents of index register.

The reference is then sent to an entity called the *Memory Management Unit* which will perform a mediated mapping to physical memory in a manner described later.

The addressing mechanism achieves the above ideal association by adding a Process-specific Segment called (again, for historical reasons) the *Linkage Segment* and having the compiler insert a local address within the Linkage Segment in the LDA instruction. (Figure 11) Local address 0 will, by convention, point to a "scratch" Segment containing local variables.
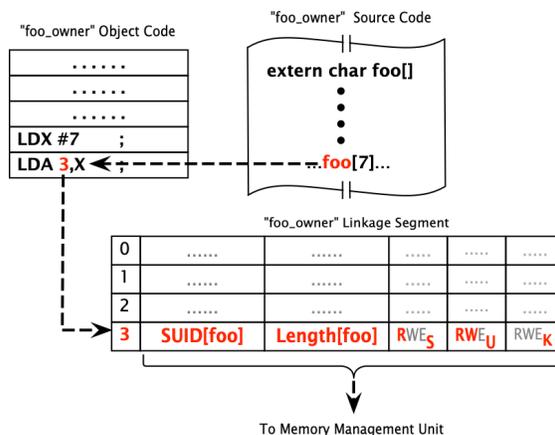

Figure 11: Descriptor in Linkage Segment

The manner in which Linkage Segments are constructed will be described below. For now it is sufficient to assume it happens correctly and consider the logic that maps a descriptor to a physical memory address.

Memory Management Unit

The Memory Management Unit, or *MMU*, is implicitly invoked whenever an address appears in one of the Instruction Layer registers. The Memory Management Unit is assumed to move segments between

9

primary and secondary memory using any of a variety of well-known techniques (paging, paged segments, fixed segments) and is not further described here. It likewise is assumed to use known caching strategies to exploit locality of reference.

Placing the MMU between the Instruction Layer and the physical memory means that it is uniformly invoked at each memory access. That invocation provides the opportunity to check the attempted instruction against the metadata contained in the Descriptor (Figure 12) before executing it. First, the bounds check is made by comparing the indexing value (Offset) to the length of "foo." Then the Layer Register is used to retrieve the relevant permissions and those are checked against the required permission of the op code; in this case a "fetch" instruction so "read" access is required. If either of those checks fail, a Trap to an error Trap Handler occurs. Otherwise, the SUID is used to index internal MMU caches of SUID/physical address pairs to locate the segment containing the byte. This physical address is then combined with an Offset to send the byte to the designated working register.
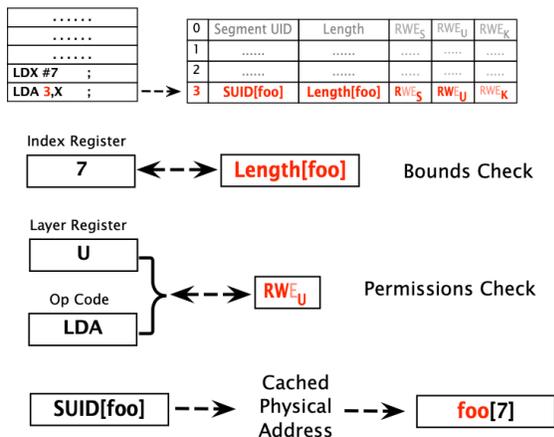

Figure 12: MMU Checks Metadata

Observations

Inherent safety of memory makes the Guard resilient against malfunctioning or malicious code at the Services Layer. That resilience would permit service code to be produced quickly by programmers or generative AI, which would then increase responsiveness to new or "zero day" attacks. Mistakes or exploits that would enable takeover of an entire platform based on unstructured memory would simply generate a Trap into an error Trap Handler.

Demand Linking

When originally formulated for the Multics [18] system, this mechanism was called "dynamic linking." That term has been pre-empted over time by a different mechanism and so the more descriptive term "demand linking" is used here.

Linking is the process of setting up the relations shown in Figure 11. In the Guard model it is deferred until the first time Process makes reference to a Segment, hence it happens "on demand."

The linking sequences begins with the translation of source to object code, which as noted above, takes place on a separate development platform. The compiler produces two segments for each unit of source code: an object code segment, which contains local addresses as described previously, and a *Linkage Segment Template* (Figure 13) which is initialized with the symbolic names of external segments. The local addresses incorporated in the object code are indexes into the template.

When a Process is initialized a copy of the Linkage Segment is made from the template and associated with the (Process, object Segment) pair. This new Linkage Segment will initially be filled with symbolic names as shown in Figure 13.

When the LDA 3,X instruction is first encountered by the Instruction Layer, that logic will fetch element 3 of the linkage segment and encounter the symbolic name "foo" instead of a Descriptor. This will force a Trap to a Utilities Layer Trap Handler which will
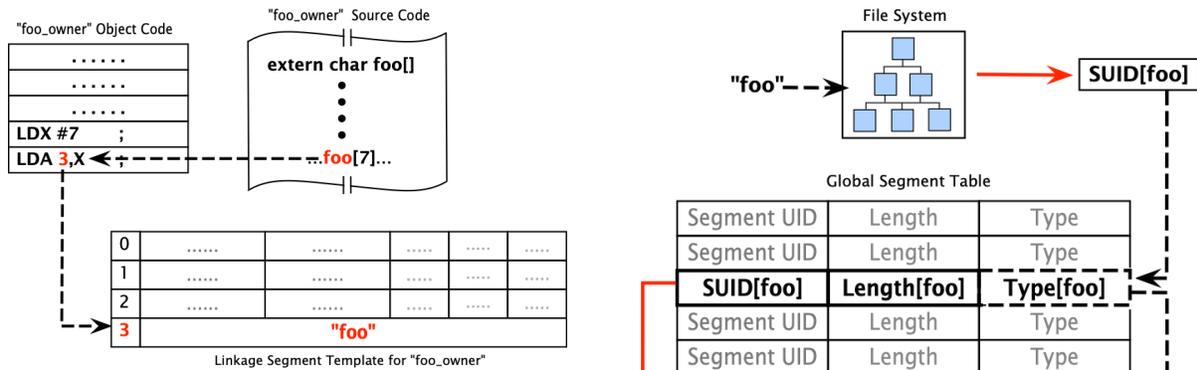
Figure 13: Linkage Segment Template.

start the sequence shown in Figure 14.

The Utilities Layer routine will use the file system to extract the Segment UID associated with the symbolic name "foo." It will then call a Kernel Layer routine which will extract a partial Descriptor for "foo" from a *Global Segment Table* or *GST*. The Segment UID and Length fields of the GST will be used to construct the first two fields of foo's Descriptor. The Type field of the GST will be used to extract the per-Layer permissions from a *Type Table* and they will complete the Descriptor, which will replace the symbolic name "foo" in the Linkage Segment and leave it as shown in Figure 11. The routines will then "unwind" back to the Services Layer, where the LDA 3,X instruction will be repeated and this time trigger an access through the MMU.

The full linkage association from symbolic name to the storage subsystem is shown in Figure 15.

Observations

The linking mechanism makes use of indirection and Traps instead of explicit procedure calls[8]. This feature, along with separate Linkage Segments for each Process, enables a Guard to update security services without interruption. Assume that a service, say "firewall" has been running for some time.If it is necessary to replace it, administrators can change its name to "oldfirewall" without affecting its operation:
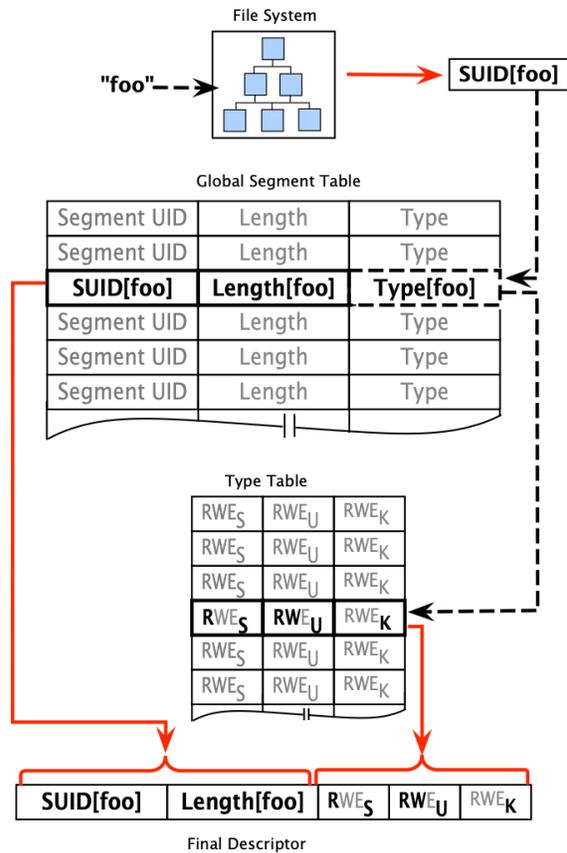

Figure 14: The Linking Process.

all linkage tests and actions have been performed. The new version then can be installed with the name "firewall" and as it executes its links will be resolved on demand. Eventually the "oldfirewall" service code will fall into disuse and it can be deleted. Such a facility will enable new and updated Services to be installed as soon as they are ready.

Demand linking also minimizes the number of completed links that are available to malicious or malfunctioning code, a common vulnerability in other linking approaches. Programmers can have a tendency to include whole libraries on a "just in case" basis. If these libraries are linked together as a large "furball" of code, vulnerability exploits can theoretically go anywhere inside that code set. Demand Link-
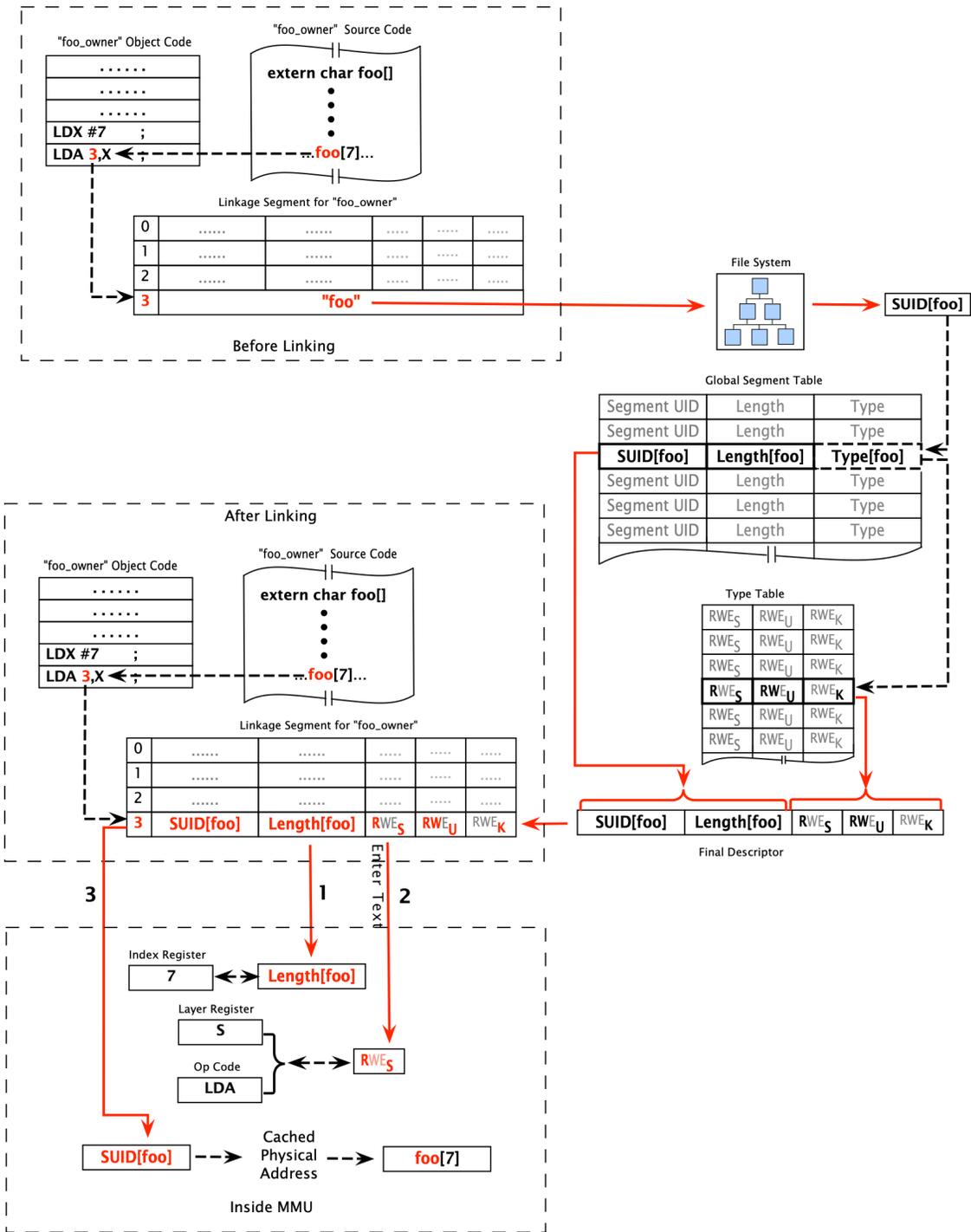
"foo_owner" Object Code

······
······
······
LDX #7   ;
LDA 3,X

"foo_owner" Source Code

extern char foo[]
•
•
•
•
...foo[7]...

Linkage Segment for "foo_owner"

0	······	·······	······	······	······
1	······	·······	······	······	······
2	······	·······	······	······	······
3		"foo"			

Before Linking

File System

SUID[foo]

Global Segment Table

Segment UID	Length	Type
Segment UID	Length	Type
SUID[foo]	Length[foo]	Type[foo]
Segment UID	Length	Type
Segment UID	Length	Type
Segment UID	Length	Type

Type Table

RWE$_S$	RWE$_U$	RWE$_K$
RWE$_S$	RWE$_U$	RWE$_K$
RWE$_S$	RWE$_U$	RWE$_K$
RWE$_S$	RWE$_U$	RWE$_K$
RWE$_S$	RWE$_U$	RWE$_K$
RWE$_S$	RWE$_U$	RWE$_K$

After Linking

"foo_owner" Object Code

······
······
······
LDX #7   ;
LDA 3,X

"foo_owner" Source Code

extern char foo[]
•
•
•
•
...foo[7]...

Linkage Segment for "foo_owner"

0	······	·······	······	······	······
1	······	·······	······	······	······
2	······	·······	······	······	······
3	SUID[foo]	Length[foo]	RWE$_S$	RWE$_U$	RWE$_K$

Enter Text

| SUID[foo] | Length[foo] | RWE$_S$ | RWE$_U$ | RWE$_K$ |

Final Descriptor

3   1   2

Inside MMU

Index Register

| 7 |

Length[foo]

Layer Register

| S |

Op Code

| LDA |

RWE$_S$

SUID[foo]

Cached Physical Address

foo[7]

Figure 15: From Source Code to Physical Storage

12

ing only links Segments that are actually referenced, and has the ability to produce event data records that can record malicious or malfunctioning code attempts to improperly access Segments.

Layer Enforcement

The Layer Enforcement mechanism supports the assurance of the Guard itself by enforcing the separation of concerns and dependency restrictions upon which that assurance is based. It is a run-time control mechanism which alters the permissions available to a Process as it moves from Layer to Layer in the course of execution.

Each of the three upper Layers has a dedicated stack. Stacks contain only Descriptor,Offset pairs; no data is pushed onto the stacks. Only Instruction Layer code is permitted to operate on stacks. This restriction insures that all memory references are mediated by the MMU, and eliminates the classic "stack overflow" vulnerability.

The distinguishing characteristic of Layer Enforcement is that permissions are both gained and relinquished as a Process moves between Layers. Properly configured, this enforces unidirectional dependency relationships which support structured assurance.

Figure 16 shows a characteristic arrangement of object code Segments surrounding the data Segment "foo" used in the earlier examples. Segment "foo" is managed by a Utilities Layer routine called "foo_owner," whose responsibility it is to validate and implement requests for changes to "foo." Segment "foo" is used by a Services Layer routine "foo_user," which can access but not modify its contents. Layers are accessible through distinguished object segments called *Gates* which have the restricted ability to change the value of the Layer Register and the stack, as well as insuring proper sequence of transition during execution of call and return instructions.

There are two Gates, one which forms the entry to the Utilities Layer from the Services Layer and a second for entering the Kernel Layer from the Utilities Layer. No Kernel Layer function is involved in the
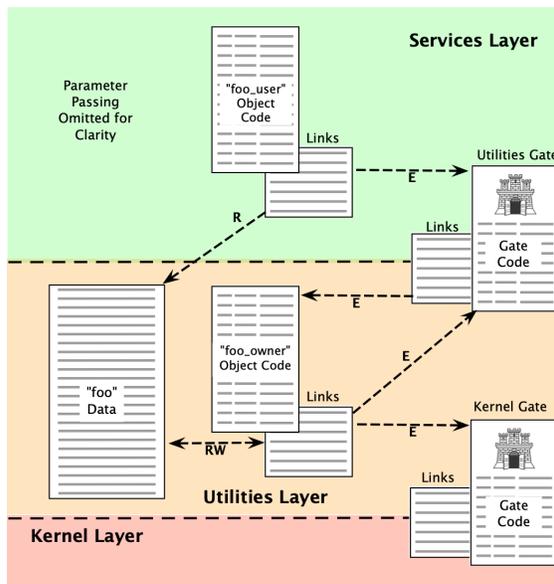

Figure 16: Layers and Gates

example and its Gate and the associated link is included only for completeness. The Instruction Layer, as noted above, is invoked implicitly by changing values of the Instruction Register.

The execution sequence of a Process requesting manipulation of "foo" begins with a call from "foo_user" to the Utilities Gate. That Gate in turn changes the value of the Layer Register and calls "foo_owner," which performs the requested operation and then returns back through the Gate to the Services Layer.

Figure 16 shows the segments and links as if all were visible to all, a circumstance that would never occur in actual operation.

The permissions when the execution point of the Process is in the Services Layer object code segment "foo_user" (shown in yellow) provide the visibility shown in Figure 17. The Segment "foo_user" is exercising its link to the Utilities Gate while the Layer Register is set to "S" for Services.

When the execution point of the Process is in the Gate shown in Figure 18, that Gate code will change
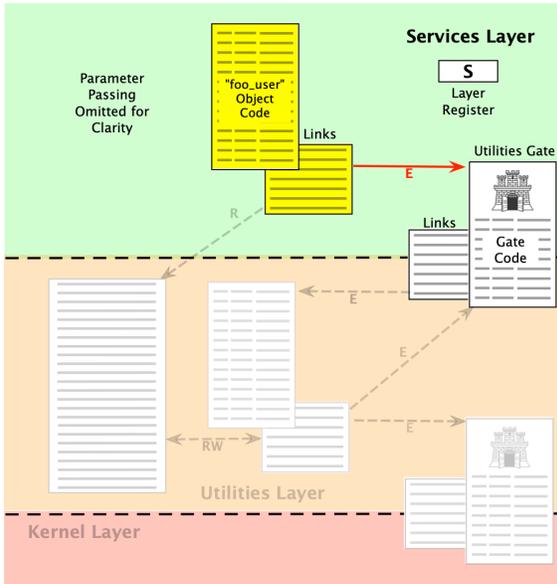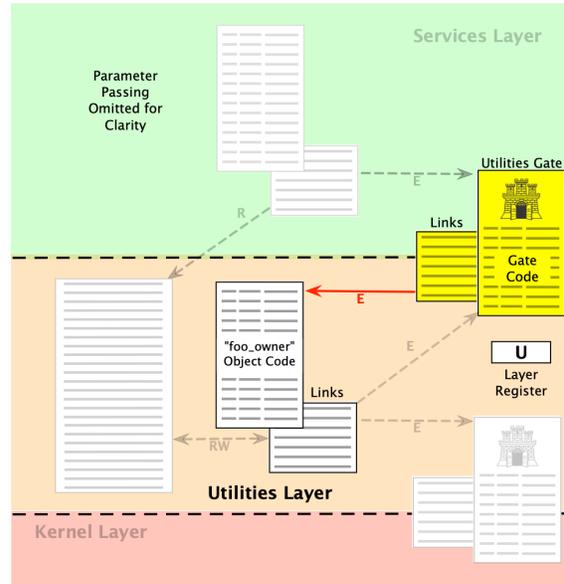
13

Figure 17: Process in Services Layer

Figure 18: Process Executing Gate

the value of the Layer Register to "U" for Utilities and execute a call to "foo_owner." At this point the Gate code is prevented from depending upon (e.g., by calling) "foo_user" or any other Services routine by the configuration of permissions, thereby preventing attacks based on maliciously or erroneously malformed parameter sets from "foo_user."

Finally, the Process executes the object code Segment "foo_owner" in the Utilities Layer, giving the accesses shown in Figure 19. At this point "foo_owner" has the option of returning back through the Utilities Gate to the Services Layer or initiating a crossing into the Kernel Layer for some restricted function, such as changing the size of "foo."

Observations

The repetitive pattern of permissions shown in the example explains why the permission fields are kept in a separate Type Table rather than entered on a per-descriptor basis in the Global Segment Table. Types are essentially equivalence class of permission, and grouping them as such in the Type Table simplifies
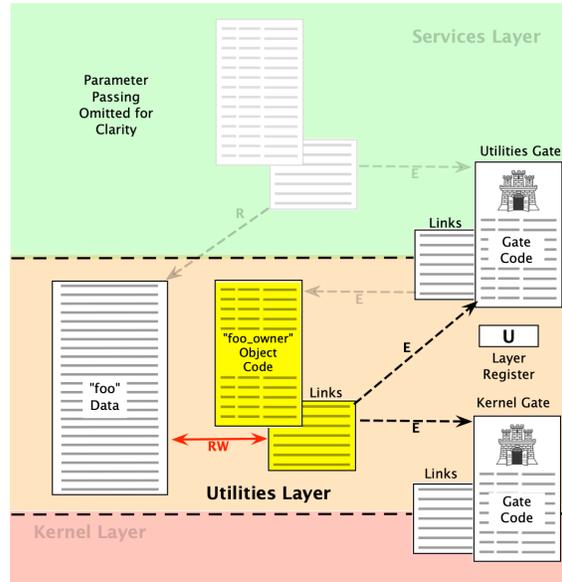

Figure 19: Process in Utilities Layer

14

an inherently error-prone configuration process and enables modifications to be made by changing a single entry rather than having to examine every descriptor in the Global Segment table to determine if the change was relevant to it.

There is an alternate mechanism in which Gates are initiated by a Trap when a link with execute permissions crosses to a new Layer, as would occur if the Segment "foo_user" were directly linked to "foo_owner." The mechanism involving explicit calling of the Gate Segment by "foo_user" was chosen in the interests of the simplest explanation of the principle that permissions are both gained and relinquished.

Notes on Further Work

It is clearly feasible for students who wish to pursue this alternative approach to do so, even to the point of attaching a running system to the internet and watching it being attacked. Inexpensive x86 machines of sufficient power are available within a student budget. Two such machines, one as a development platform and the other as a target would be enough to support even a team of students. Filling in the omissions in the model presented here with executable code would provide experience in machine-level programming, integration, and assurance in a structured program with defined goals.

A logical plan would be to first construct a virtual machine [28] beginning with a Memory Management Unit and then an Instruction Layer to provide a suitable register and command set[9]. The next step would be to adapt a suitable compiler to generate Linkage Segments as well as code for the virtual machine. This could be tested with hand-generated Descriptors until Demand Linking was built. The last step in producing a basic platform would be Layer Enforcement.

Once a platform had been built the students could gain experience in vulnerability assessment by subjecting it to penetration tests and eventually connecting it to the internet to observe real-world attacks.

Acknowledgements

References

[1] A. Zonnenshain and S. Stauber, "The Many Faces of Systems Engineering," In *Proc. 14th International Council on Systems Engineering (INCOSE) Symposium* 2104, pp 923-937.

[2] "Concepts of Operations (CONOPS) for Systems Engineers," Accessed December 3, 2024. `https://reqi.io/articles/concept-of-operations-conops`

[3] J. M. Griffin and K. Mei, "How Do Crypto Flows Finance Slavery? The Economics of Pig Butchering" Accessed December 10, 2024. `http://dx.doi.org/10.2139/ssrn.4742235`

[4] Boebert, W. E., "A Survey of Challenges in Attribution." In *Proceedings of a Workshop on Deterring Cyberattacks* pp. 41-52. Washington, DC: National Academies Press, 2011.

[5] L.A. Belady and M.M. Lehman, "A model of large program development" In *IBM Systems Journal* No. 3, July 1976, pp 225-252.

[6] F. McKee and D. Noever, (2023) "Chatbots in a Honeypot World" arXiv preprint arXiv:2301.03771

[7] Forrester Research Inc. "Developing a Framework to Improve Critical Infrastructure Cybersecurity" National Institute of Standards and Technology, April 8, 2013.

[8] A. Rosenblueth and N. Wiener, "The Role of Models in Science", *Philosophy of Science*, Vol. 12, No. 4 (Oct., 1945), pp. 316-321

[9] A.M. Turing, "The Chemical Basis of Morphogenesis",*Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, Vol. 237, No. 641. (Aug. 14, 1952), pp. 37-72.

[10] F.M.Brooks, "The Tar Pit," In: *The Mythical Man-Month*, Addison-Wesley, 1975.

[11] E.W. Dijkstra, "Structure of the 'THE' - Multiprogramming System" In *Communications of the ACM* V II, No. 5, May 1968 pp 34-346

[12] B. De Win, F. Piessens , W. Joosen, T. Verhanneman. "On the importance of the separation-of-concerns principle in secure software engineering." In *Workshop on the Application of Engineering Principles to System Security Design* ACSA November 2002 (pp. 1-10).

[13] D.L. Parnas, "On a 'Buzzword': Hierarchical Structure." In: Gries, D. (eds) *Programming Methodology. Texts and Monographs in Computer Science.* Springer, New York, NY. 1978

[14] "Failure Mode and Effects Analysis." Accessed December 3, 2024. `https://asq.org/quality-resources/fmea`

[15] "Fault Tree Analysis (FTA): definition, applications and benefits." Accessed December 3, 2024. `https://blog.infraspeak.com/fault-tree-analysis-fta/`

[16] W.E. Boebert, T.R. Markham, R.A. Olmstead, "Data Enclave and Trusted Path System," U.S. Patent US-5276735-A

[17] C.W. Johnson, "What are Emergent Properties and How Do They Affect the Engineering of Complex Systems?" (2005) Accessed 17 December 2024. `https://www.dcs.gla.ac.uk/~johnson/papers/RESS/Complexity_Emergence_Editorial.pdf`

[18] "Multics." Accessed December 3, 2024. `https://multicians.org/index.html`

[19] L.J. Fraim,"Scomp: A Solution to the Multilevel Security Problem,"*IEEE Computer Magazine*, July 1983, pp. 26-34

[20] P.G. Neumann and R.J. Feiertag, "PSOS Revisited", In *Proc. of the 19th Annual Computer Security Applications Conference*, 2003

[21] W.E. Boebert, R.Y. Kaln, W.D. Young, S.A. Hansohn, "Secure Ada Target: Issues, Systems Design, and Verification" In *Proc. 1985 IEEE Symposium on Security and Privacy*, pp 176-183

[22] R.E. Smith, "Cost Profile of a Highly Assured, Secure Operating System" In *ACM Transactions on Systems and Information Security* V 4, No. 1, Feb. 2001 pp 72-101.

[23] "The Origin of Sidewinder." Accessed January 11, 2025. `https://web.archive.org/web/20020627134527/http://www.securecomputing.com/index.cfm?sKey=1024`

[24] "Heisenbug" Accessed January 10, 2025. `https://wordspy.com/words/heisenbug/`

[25] R. Niba. "Understanding Daemons: Their Role and Function in Computing" Accessed December 30, 2024. urlhttps://www.bioscentral.com/understanding-daemons-their-role-and-function-in-computing/

[26] E. A. Feustel, "On The Advantages Of Tagged Architecture" In *IEEE Transactions on Computers*, Vol. C-22, No. 7, July 1973, pp 644-656

[27] "What is memory safety and why does it matter?" Accessed December 3, 2024. `https://www.memorysafety.org/docs/memory-safety/`

[28] I. Ali and N. Meghanathan, "Virtual Machines and Networks – Installation, Performance, Study, Advantages AND Virtualization Options" In *International Journal of Network Security & Its Applications*, Vol.3, No.1, January 2011, pp 1-15. DOI : 10.5121/ijnsa.2011.3101

[29] A. Badhoutiya, Z. Jaffer, H. M. Hussein, A. Juyal, M. Mittal and R. Anand, "Field Programmable Gate Array: An Extensive Review, Recent Trends, Challenges and Applications," In *Proc. 2024 11th International Conference on CoBmputing for Sustainable Global Development (INDIACom)* February 2024.

Notes

[1] Previously: Honeywell, Secure Computing Corporation, Sandia National Laboratories. Author contact: boebert@swcp.com

[2] "What You See Is What You Get," originally a light-hearted way to describe word processors that displayed formatted instead of raw text as you typed.

[3] Sources include Multics [18], Scomp [19], PSOS [20], SAT [21], LOCK [22], and Sidewinder [23] The contributions of too many individuals to list are hereby acknowledged; this model is a synthesis and no claim of invention is made or should be implied.

[4] The situation is worse if the interleaving is determined by indeterminate external events. Such mechanisms can exhibit the phenomenon called "Heisenbugs," [24] where errors manifest themselves in operational use but vanish when instrumentation code is inserted to look for them.

[5] Also known as "unprogrammed transfers" or "internal interrupts."

[6] Because their operation is hidden from programmers or analysts reading code, Traps and Trap Handlers should be used sparingly and with care.

[7] The assumption here is that the Instruction Layer is eventually embodied in hardware

[8] Indirection and Traps carry a performance penalty, which historically has inhibited their acceptance in application system. This inhibition does not apply to a dedicated security system.

[9] If the results of the effort looked promising, elements such as a MMU coprocessor could be made with a a field programmable gate array [29].