Abstract. We introduce the N-variant systems framework for protecting vulnerable services from attack. Our approach employs artificial diversity techniques. However, instead of relying on keeping secrets like previous approaches, we use an architectural framework to provide a high degree of assurance without needing any secrets. By designing variants that satisfy disjoint sets of possible attack preconditions, and ensuring that an attacker must compromise all variants simultaneously, we can obtain provable security against classes of attacks. In this paper, we present the N-variant systems framework, identify types of variation that may be used to provide immunity against particular attack classes, describe a prototype implementation of a simple example of an N-variant system based on partitioning the space of valid addresses, and discuss issues involved in making our approach secure and practical for complex servers.

1 Introduction

Artificial diversity has been proposed as a mechanism for defending computer systems against attack and several techniques for producing diversity have been developed including memory address space randomization [5, 26] and instruction set randomization [4, 17]. These mechanisms provide security in two ways: (1) in protecting a network of systems, they provide diversity in that the same exploit will not succeed against different machines on the network since they are randomized differently; and (2) in protecting a single system, they provide security through obscurity since the attacker does not know the memory layout or instruction set. Although diversity defenses are very effective against existing attacks, implementations of both memory address space randomization [23] and instruction set randomization [25] have been demonstrated to be vulnerable in certain circumstances.

The problem is that previous diversity defenses rely on keeping secrets, and keeping secrets is hard. A remote attacker can determine the randomization...
key using an incremental guessing or probing attack. In addition, if security depends on keeping a key secret, that key could be disclosed through other channels. Although the particular attack avenues identified so far can be thwarted by particular modifications to the defenses, any system security mechanism that is based on keeping a secret is vulnerable to a secret compromise attack. In this paper, we introduce a new approach to security systems using artificial diversity but without relying on secrets.

Figure 1 illustrates our architectural framework. The original server process $P$ is replaced with two variants, $P_1$ and $P_2$. The variants maintain the semantics of $P$ on normal inputs. They are, however, artificially diverse in a way that makes them behave differently when given abnormal inputs in the form of an attack. The key insight behind our proposed work is that in order for an attacker to exploit a vulnerability in $P$, the same pathway must be available to exploit the vulnerability in both $P_1$ and $P_2$. It may be possible to construct an attack that succeeds against one of the variants; but, as long as the same input cannot compromise both variants simultaneously, the monitor will detect the anomalous behavior and block any output from reaching the attacker. After detecting the attack, the monitor can restart the server in a known uncompromised state. If we can prove the set of attacks that could succeed against $P_1$ is disjoint from the set of attacks that could succeed against $P_2$, then overall security is assured if the input replicator and monitor can be implemented correctly. We will not be able to claim this for all possible attacks, of course, but for a well defined class of attacks we may be able to prove that no instance of an attack in the class can succeed against both variants.

As a simple example, suppose $P_1$ and $P_2$ use disjoint memory spaces. Any absolute memory address that is valid in $P_1$ is invalid in $P_2$, and vice versa. Since the legitimate programs are transformed to provide the same semantics regardless of the memory space used, the behavior on all normal inputs is identical. However, if an exploit uses an absolute memory address, it is guar-
anteed that that address must be invalid in one of the two variants. A monitor process observes the two variants, and only releases outputs back to the client when both variants successfully process an input. If the monitor observes discrepancies between the variants, it restarts the processes in known uncompromised states.

We refer to the general framework shown in Figure 1 as an N-variant system. The framework generalizes to having more than two variants, but for most security properties two variants are sufficient. The concept is inspired by, but fundamentally different from, the technique known as N-version programming [2, 10, 16]. N-version programming involves the creation of N different implementations of a specification by N independent development groups. The hope is that separate development will yield versions with different faults. In an N-variant system, program variants are created by mechanical transformation and they are engineered specifically to enable attack detection. Furthermore, the N-variant system’s output is regulated on the basis of the monitor’s observation of attacks rather than on the basis of output voting.

Our approach results in a system with valuable properties that cannot be achieved with previous approaches:

• No secrets – we do not need to assume any secrets can be kept from adversaries in order to achieve our security properties.
• Detection – we can detect a large class of security attacks in real-time.
• Provable immunity – our approach has the potential to enable proofs that a system cannot be compromised by a class of attacks.

Next, we describe our approach to hiding vulnerabilities by producing variants that satisfy disjoint precondition sets. Section 3 illustrates our approach by describing a prototype implementation. Section 4 discusses issues involved in using our approach to establish valuable security properties for real systems.

2 Vulnerabilities and Variants

We assume a server program containing some vulnerability. For now, we focus on vulnerabilities that allow attackers to corrupt memory or hijack control flow. Examples include format string vulnerabilities that may allow an attacker to read or write data at an arbitrary absolute memory address [24, 22] and stack buffer overflow vulnerabilities that may allow an attacker to inject code into a running process and jump to it [1, 13].

In order to attack the system, an attack must create an exploit that uses the server vulnerability to achieve some desired malicious result. A successful attack depends on the state of the server process satisfying a set of preconditions needed by the attack. For example, the attack may rely on the access
control variable being stored at a particular location. Program transformation techniques can be used to alter properties of the server process to change its state in ways that prevent it from satisfying attack preconditions. For example, memory address space randomization could make the location of the access control variable unpredictable. However, if the attacker learns the new location, a different attack could be constructed with a new precondition that is satisfied by the variant.

Our goal is to develop transformations (possibly in combination with runtime checks) such that two variants, \( P_1 \) and \( P_2 \), can be created with the following property: no attack of a given type can be constructed for which its necessary preconditions are satisfied by both variants. For the illustrative example in which the attack depends on the precondition of the access control variable being stored at a particular location, if the variable is stored at different locations in \( P_1 \) and \( P_2 \), any attack that depends on a precondition that involves the location of the variable cannot be satisfied simultaneously by both processes.

The success of our approach depends on our ability to construct sets of variants where the intersection of the sets of precondition-satisfying exploits for each variant is empty. Next, we describe four types of variations we can use to disjoint precondition sets.

**Memory Organization.** There are many possible ways we could make variants in which classes of memory accesses that are valid in one variant are always invalid in another. The simplest idea is to simply partition the address space between the two variants. This could be done at the segment level using a modified loader that provides one variant with access to memory segments whose high-order bit is 0, and provides the other variant with access to memory segments whose high-order bit is 1. This approach is easy to implement, and has the nice property that no monitoring is required beyond what is already provided by the operating system. All addresses that are valid in one variant will cause an illegal address fault in the other variant. If an exploit depends on accessing an absolute address, it will necessarily crash one of the variants that will be noticed by the monitor.

This simple approach does not prevent all memory corruption attacks, however, since some attacks do not depend on accessing an absolute memory location. For example, a typical stack smashing buffer overflow attack works by overwriting a buffer on the stack to replace the original return address offset with a new value that causes control to jump to a location in the vulnerable buffer that contains the attacker’s injected code. Disjoint memory spaces fail to prevent this attack since on most processors return addresses offsets are within segments, not absolute addresses. Hence, the actual precondition for this attack is being able to overwrite the return address on the stack with an offset that will reach a location in the vulnerable buffer. To generate an unsatisfiable constraint for this precondition, we need to ensure that any relative
jump that is valid in one variant is guaranteed to be invalid in the other variant. Achieving this property requires generating more complex artificial constraints on the variants than the loader example above, but could be done efficiently with a compiler or binary rewriter. One approach would be to use alignment of the low-order bits instead. For one variant, the compiler generates code such that the offset between every call site and target is even, and in the other variant the compiler generates code such that it is odd. When the variants execute, a run-time monitor checks that all return jumps satisfy the appropriate alignment property.

**Instruction Set.** A precondition for a code injection attack is that a particular sequence of bits produce a particular behavior on the target processor. For example, an attack may have a precondition that \(0xeb\) corresponds to a jump opcode. Instruction set randomization attempts to foil attacks by breaking attack preconditions regarding the instruction set. It relies on keeping the randomization key secret, since the original jump instruction is still available if the attacker can determine the corresponding randomized instruction. With an N-variant system, we can avoid this possibility, by constructing variants with disjoint instruction sets.

If we consider a class of code injection attacks that involve at least one control flow changing instruction (e.g., a jump, conditional jump, return or call) in the injected code, we can construct an unsatisfiable constraint by ensuring that all sequences of bits that are control flow changing in one variant are guaranteed to be invalid instructions on the other variant. The variants would be created using a compiler or binary rewriter to use only the available control instructions. Note that secrets are no longer necessary: we can publish both instruction sets openly, since even if the attacker can generate effective code for one variant it is guaranteed to crash the other variant. The variants must run in emulators that allow only a subset of control flow instructions. Using dynamic translation techniques, such an emulator can be implemented with little overhead [3, 21].

**Scheduler.** Race condition attacks depend on events if different threads occurring in a particular order. For example, a typical time-of-check time-of-use exploit depends on an event that modifies a resource (often the symbolic link associated with some file descriptor) happening in one thread between events in another thread that check access rights to that resource and use the resource. The race condition attack succeeds if the attacker is able to create a situation where the access check passes for a given descriptor, the other thread changes the resource associated with that descriptor to one inaccessible to the attacker, and then the first thread continues using the original descriptor but
manipulating the protected resource. This suggests that schedule variations could be used to create unsatisfiable preconditions for race condition attacks. The goal is to ensure that the variants do not have the same event orderings. One approach would be to have variants with different schedulers guaranteed to have different interleaving properties. For example, one scheduler could always schedule the lowest thread ID to run first, while the other scheduler gives priority to the highest thread ID. The challenge is to guarantee that the same event ordering is not possible in both variants.

**File Naming.** Attacks that depend on reading or writing to particular files have a precondition that they are able to name the target file. We can make that unsatisfiable by ensuring that “X is a valid filename” is an unsatisfiable constrain across a set of variants for all possible values of X. There are two possible approaches: one is to have the variants each maintain their own file system, with disjoint name space; the other is to have variants share a common file system, but interact with the file system using a proxy that transforms names differently for each variant. This would provide immunity from any attack with a precondition involving naming a specific file.

3 Example

To evaluate our approach, we built a prototype implementation and demonstrated its effectiveness defending a simple server from an attack. In Section 4, we discuss some of the technical issues that need to be solved before we can use it to protect a complex server. Our prototype implementation uses variants with disjoint memory spaces to thwart attacks, such as exploits of format string vulnerabilities, that depend on absolute memory addresses. We built our prototype on FreeBSD 5.3, but all the features we used are also available on Linux.

3.1 Prototype Implementation

Our prototype implementation follows the framework introduced in Figure 1. Both the input replicator and monitor are implemented by a single process we call nvpd. Although our test server is simple, in order for our approach to work in practice it is essential that modification to the server source code is necessary. We impose the same constraint on our prototype implementation. Hence, it is necessary for each server variant to execute in a context where it appears to be interacting directly with the client as it would normally. We accomplish this using FreeBSD’s divert sockets which allow low-
level access to the IP stack (Linux implementations now also provide similar functionality).

**Input replicator.** We use ipfw, a firewall implementation for FreeBSD [15] with a rule that redirects packets on port 80 (HTTP server) to our nvpd process. The nvpd uses divert sockets to manipulate incoming IP packets. Each incoming packet is sent to both server variants. To preserve the variant’s illusion that it is dealing directly with a normal client, nvpd must adjust the TCP sequence numbers to be consistent with the variant’s numbering. In our implementation, we use the original client sequence numbers in the diverted socket with the $P_1$ variant, and rewrite packets sent to $P_2$ to be consistent with its expected sequence numbers.

**Variants.** The server variants differ in the locations of their data segments. The memory addresses used by $P_1$ and $P_2$ are disjoint: any data address that is valid for $P_1$ is invalid for $P_2$, and vice versa. We accomplish this using the --section-start linker option. We use the same source code for both variants, but compile the $P_1$ variant using --section-start,.data=0x44444444 and the $P_2$ variant using --section-start,.data=0x55555555. This ensures that their data segments start at different locations and their sets of valid data memory addresses are disjoint.

**Monitor.** Instead of sending responses directly to the client, the server variant responses will be diverted back to nvpd. The nvpd process will buffer the responses from both variants. The responses from $P_1$ are transmitted back to the client only if a comparably long response is received from $P_2$. Hence, if $P_2$ crashes on a client input, the response from $P_1$ will never be sent back to the client. For a normal request, both $P_1$ and $P_2$ will respond with the same length packet. Once it has received both responses, nvpd will send the $P_1$ response back to the client. For an attack request, one of $P_1$ or $P_2$ will crash and nvpd will not send any response back to the client.

### 3.2 Server

To test our approach, we implemented a simple HTTP server, vulnerabilitid. It handles a single GET or POST request from a client. The server parses the HTTP request and imports several of its fields as environment variables and then displays them back to the client. This is analogous to the construction of a process environment that is typical for external CGI and server scripts. The server has an internal Boolean variable, trusted, that indicates whether or not the client is an authenticated user.

Our server implementation contains a seeded format string vulnerability in the code that displays the environment:
for (char **p = environ; *p != NULL; p++) {
    char buf[256];
    snprintf (buf, sizeof buf, *p); buf[(sizeof buf) - 1] = '\0';
    fprintf (stdout, "%s\n", buf);
}

The problem with the above code is with the use of the environment variable data directly without a format-string. Since the untrusted client controls the value of these string, the value of *p could include format codes that an attacker can exploit to access memory arbitrarily. To avoid this problem, the snprintf call should have been written as snprintf (buf, sizeof buf, "%s", *p) to input the user data as a string.

3.3 Exploit

To exploit the seeded server vulnerability, an attacker needs to determine the location of the trusted variable in memory and construct an input URL string that will write a non-zero value into that location. An attacker who knows enough about the server may be able to determine this location analytically; otherwise, it can be found by using the string format vulnerability to probe memory. Although this is a contrived vulnerability, similar problems have been found in widely-used servers.

For our $P_1$ server, trusted is stored at address 0x44444455 which encodes as the character sequence UDDD. The printf format code %n format code gives us the means to write an integer value to a specific address: it stores the number of bytes printed so far in the corresponding parameter. Normally, this address is given as an argument to the function in the source code when writing the program. In our case, we use the fact that the format string is under our control to pop an arbitrary number of arguments off of the execution stack until we begin to see the format string itself. To find the format string on the stack we query our server with a URL with a URI portion that begins with /___DDDD and then keep appending %p tokens until we see a value of 0x44444444 displayed at the end the string in the server output. This corresponds to the DDDD in the attack input. The final step of the attack is to replace the DDDD in the URI with the location of trusted, UDDD, and to replace the final %p with a %n. Requesting this new URL causes the non-zero number of bytes written so far to be stored in the trusted variable. If trusted controls access to critical resources, changing its value compromises security.

3.4 Thwarting Attacks

The attack described in the previous section works against $P_1$; a similar attack with a different target location would work against $P_2$. Both $P_1$ and $P_2$
are could be compromised independently, but the memory locations needed to exploit them are different. Hence, they cannot be compromised simultaneously. Within our N-variant system architecture, there is no version of the exploit that works, however. If the attacker constructs an exploit using the address of data in $P_1$ it must use an address that is invalid in $P_2$: $P_1$ is successfully compromised, but $P_2$ crashes when it attempts to write to the invalid address; if the exploit uses the address of data in $P_2$ the address used must be invalid in $P_1$. We have obtained the desired property: any exploit that would succeed against one of the variants causes the other variant to crash. This is noticed by the monitor and no packets from the compromised server are transmitted to the attack client. The Appendix shows the details of the interactions between the client, nvpd and variants.

4 Discussion

Our prototype implementation illustrates the potential for N-variant systems to protect vulnerable servers from large classes of attacks. In order for our approach to provide strong security guarantees while protecting complex servers without changing the server semantics on normal requests, a number of issues remain.

Provable security properties. Although the cryptography community has developed techniques for proving security properties of encryption algorithms and cryptographic protocols, similar assurance of system security properties remains an elusive goal. System software is too complex to prove it has no vulnerabilities, even when only a particular class of vulnerabilities is considered. Previous techniques for thwarting exploits of vulnerabilities have used ad hoc arguments and tests to support claimed security properties. Clever attackers, however, regularly find ways to successfully attack such techniques [8]. The N-variant systems approach offers the promise of a more formal security argument. If we can prove that the artificial diversity produces variants such that the intersection of the sets of attack preconditions that can be satisfied by each variant is empty, then we can be assured that no attack in a class depending on those preconditions could be successful. The soundness of the argument depends on correct behavior of both the input replicator and monitor. Since these can be relatively simple, this may be feasible.

Persistent state. Our experimental target server does not interact with any external state, so there is no problem with having both $P_1$ and $P_2$ handle each request. A typical web application, however, interacts with external state such as a file system and database. For example, a client request may cause $P$ to
insert a record in a database. If \( P_1 \) and \( P_2 \) both share the same database and insert the record, the record will be inserted twice and the semantics of the N-variant system will not be consistent with the semantics of the original application.

One possible solution is to replicate all state used by \( P \) so that both \( P_1 \) and \( P_2 \) have their own copy of the external state. For applications that use a large database, this could be very expensive. Further, if the persistent state is external and may be modified by other applications when \( P \) is running, keeping both copies of the state consistent may be quite difficult.

A more practical solution is to extend the monitor to intercept calls that involve external state. As with client responses, the call would not proceed until both variants make the same request. The wrapper could then make the request once, and return the response to both variants. Several suitable techniques are available for efficiently wrapping calls [9, 14].

**Nondeterministic behavior.** Our prototype monitor assumes that both variants must produce the same size response when they are handling a legitimate request. For our simple target server, this is the case. The responses a complex server generates, however, may depend unpredictably on factors like the system clock, pseudorandom number generator, process ID and thread interleaving. We can deal with some of these issues the same way as we propose to handle persistent state above – intercept the nondeterministic functions and implement wrappers to provide both variants with the same responses. For thread interleaving, providing a consistent context would require fine grain control of the scheduler for each variant. Doing this increases the complexity of the monitor, and hence, the likelihood it contains a vulnerability.

An alternate approach would be to develop techniques for tolerating some amount of differences between the states of the variants. This would make proving security properties considerably more difficult in some cases, since proofs can no longer depend on the variants starting in equivalent states at the beginning of processing a request. Once we allow such differences, we need to worry about multi-stage attacks in which an attacker generates a sequence of requests to get both variants in different states in which one of the variants is compromised but no single request needed an unsatisfiable precondition.

**Cost.** Our approach appears to require substantial resources – it more than doubles the load on the server since it requires running two versions of \( P \) in addition to the input replicator and monitor. The actual cost may be reduced if we use the wrapping system calls approach so that both variants can share a single database of file system call. For server applications whose performance is primarily IO-bound, the variant processes may not reduce throughput
so much. On the other hand, the time system administrators spend dealing
with successful security compromises is far more expensive than processing
power these days. For many services, even a doubling of server load may be
a small price to pay for the benefits of provable security against large classes
of attacks. Further, in practice many systems already employ replication to
provide improved availability. Replicates usually involved complete copies
of system state so that failure in one replicate does not lead to loss of data. For
systems such as these, our approach adds little additional cost.

**Recovery.** The monitor detects a possible attack by observing different re-
sponses from the variants and inconsistencies in their internal states. In our
example implementation, it is enough to notice the different response sizes
and that one of the variant processes has crashed. At this point, one of the
variants is in a possibly compromised state and the other one has crashed.
After detecting the attack, the monitor needs to restart the service in an un-
compromised state. For a simple stateless server like our example, the moni-
tor can just restart both variant processes. For a stateful server, recovery is
more difficult. One approach is to maintain a third variant that keeps track of
known uncompromised state and can be used to restart the other two variants
after an attack is detected. The third variant could be the original $P$, except it
would be kept behind the state of the other two variants. The input replicator
would delay sending input to the third variant until after both $P_1$ and $P_2$ have
processed it successfully. This ensures that the third variant only sees non-
attack input and can be used to recover an uncompromised state after an at-
tack is detected.

5 Related Work

The use of N-version programming to help with system security was pro-
posed by Joseph [16]. That research analyzed design diversity as manifested
in N-version programming to see whether it could defeat certain security at-
tacks. The authors developed an analogy between faults in computing sys-
tems that might affect reliability and vulnerabilities in computer systems that
might affect security and argued that N-version programming techniques
might allow vulnerabilities to be masked.

A major problem with N-version programming is the high cost of duplicat-
ing human resources throughout the development process. A related form of
diversity in which semantics-preserving transformations are applied automati-
cally to software specifically to try to reduce the impact of security vulner-
abilities has been proposed by several authors [4, 5, 17, 18, 26]. However, de-
randomization attacks have recently been presented against several of these
diversity techniques [23, 25]. A major advantage of our N-Variant Systems approach is that we do not rely on secrets for our security properties.

Work in the distributed systems community has used active replication to achieve fault tolerance [6, 7, 11, 12, 19]. With active replication, all replicates are running the same software and process the same requests. Unlike our approach, however, it does nothing to hide design flaws in the software since all replicas are running the same software. To mitigate this problem, Schneider and Zhou have suggested proactive diversity, i.e., periodically randomizing replicas, to justify the assumption that server replicas fail independently and to limit the window of vulnerability in which replicas are susceptible to the same exploit [20]. Active replication and N-Variant Systems are complementary approaches. Combining them can provide the benefits of both approaches with the overhead and costs associated with either approach independently. Many of the problems we discussed in Section 4 including dealing with persistent state, nondeterminism and recovery have been addressed in active replication systems [7] and aspects of their solutions can be applied to N-variant systems.

6 Conclusion

The N-variant systems approach offers a new way to protect vulnerable servers from attacks. We have illustrated our approach with a prototype implementation that protects a vulnerable server from exploits that depend on referencing absolute addresses. By considering different types of artificial diversity, our approach can be generalized to large classes of important attacks including code injection and memory corruption attacks. N-variant systems offer the promise of provable assurance of resilience against important classes of attack.
References

Appendix: Interactions

Figure 2 shows the interactions between the client, nvp\textsuperscript{d}, and server variants for a typical session. Most of the packets in this brief TCP connection are involved in the connection set up (the initial 3-way handshake of SYN\text{s} and AKC\text{s}) or the connection closing (the FIN\text{s} and ACK\text{s} at the end). Note that nvp\textsuperscript{d} must adjust the sequence numbers of packets between the client and variants to maintain the illusion of a normal TCP connection.

The packet labeled \#1 contains a safe HTTP request, GET / HTTP/1.1, as the payload. \textit{P$_1$} responds initially with packet \#2, a response that only contains part of the HTTP response header (HTTP/1.0 200 OK). \textit{P$_1$} happens to be the first to send the remainder of the response in packet \#3, but it is not forwarded to the client by nvp\textsuperscript{d} since \textit{P$_2$} has not yet sent packet \#4. When \textit{P$_2$} does, nvp\textsuperscript{d} forwards \textit{P$_1$}’s packet to the client as packet \#5. \textit{P$_2$} happens to be the first to retransmit the remainder of the response with packet \#6, and it is recorded by nvp\textsuperscript{d}, but the reply, packet \#10, is not sent to the client until after nvp\textsuperscript{d} receives packet \#9 from \textit{P$_1$}. Note that \textit{P$_2$} keeps retransmitting the response (packet \#8) until it receives the ACK from the client via nvp\textsuperscript{d}, just like a normal TCP connection. Packet \#7, the client ACK’s the initial reply, is just resent to both \textit{P$_1$} and \textit{P$_2$}. 

14
When the request is an attack URL as described in Section 3.3, one of $P_1$ or $P_2$ will crash when generating output. The packet flow to the client is unchanged through packet #7. Packet #8 and #9, however, will now be different lengths since the crashed server will not return a full response. Since the crashed can by never send more data to match the length of the other packet, packet #10, will never be sent to the client, thereby averting disclosure of any information by the compromised server. The monitor will observe the process crash for $P_1$ and restart the variants in known uncompromised states.