LOBOTOMY
An Architecture for JIT Spraying Mitigation

Martin Jauernig
Vienna University of Technology

Matthias Neugschwandtner
Vienna University of Technology

Christian Platzer
Vienna University of Technology

Paolo Milani Comparetti
Lastline Inc.

***Abstract*—JIT spraying has an assured spot in an attacker's toolkit for Web browser exploitation: With JIT spraying an attacker is able to circumvent even the most sophisticated defense strategies against code injection, including address space layout randomization (ASLR), data execution prevention (DEP) and stack canaries.**

In this paper, we present LOBOTOMY, an architecture for building injection-safe JIT engines. LOBOTOMY is secure by design: it separates compiler and executor of a JIT engine in different processes that share the memory regions containing the compiled code. This allows us to use least-privilege access rights for both processes, preventing memory regions to be mapped with write- and execute-rights at the same time.

Our proof-of-concept implementation that modifies the well-known Firefox JIT engine Tracemonkey shows both the effectiveness and real-world feasibility of our architecture. Additionally, we provide a thorough evaluation of our version compared to an unmodified baseline and competing approaches.

I. INTRODUCTION

Virtual machines that process interpreted code often employ just-in-time (JIT) compilation in order to increase the performance of frequently used parts of a program. While the most prominent example for JIT compilation is the Javascript engine of modern browsers, it is also used for e.g. Adobe's Actionscript.

Unfortunately, the widespread use of JIT compilation has also made it an attractive target for (browser) exploitation. So-called *JIT spraying* is an attack vector that was first described by Blazakis et al. in [3]. The 2012 Pwn2Own contest [6] has impressively shown the potential of JIT spraying attacks, when Firefox was compromised by a vulnerability that enabled such an attack. In a nutshell, the typical procedure is to first use the JIT engine to produce native shellcode and then leveraging a bug in the engine's compiler component to redirect the control flow. To generate shellcode by means of a JIT engine, it is fed with an input program that consists of numerous operations on large constant operands, such as XOR. When being executed from the right offset, these constant operands decode to the malicious instructions that constitute the shellcode.

At the root of JIT spraying are memory regions that are both write- and executable within the address space of a process, the JIT engine. To counter this attack, one has to find a way to separate the write permissions from the execute permissions, either temporally or spatially. One approach is to allocate two regions of memory backed by the same temporary file, separated by a certain offset within the same process. This is a stable and fast solution. However, should an attacker discover this (not explicitly randomized) offset, any security gained by implementing such a mechanism would be lost again [10]. Another possible approach is to change the mapping on demand: A memory region would be mapped writable and only marked as executable when the compiled code needs to be run [4]. However, if an attacker could stall, delay or lengthen the execution of compiled code while the associated pages are marked executable, a successful exploit is still feasible. Additionally, this approach may leave memory regions unprotected that contain compiled code but are not currently being executed.

In this paper, we present LOBOTOMY, a novel, generic approach to harden JIT engines against code injection. Instead of separating writable and executable memory by time or by an offset within the memory space of a process, the two regions are accessed from two different processes: compiler and executor. As a result, an attacker exploiting the compiler has no access to executable codepages. Essentially LOBOTOMY re-establishes the conceptional separation of data and code that was lost with the introduction of JIT compilation.

To summarize, our contributions are threefold:

- We introduce LOBOTOMY, a novel architecture for JIT engines to improve their resilience against JIT spraying.
- We present a proof-of-concept implementation of this architecture using Tracemonkey as it is shipped with Firefox 5.
- We evaluate our approach and the impact this architecture has on the JIT engine and compare it with simpler protection mechanisms.

II. MOTIVATION AND PROBLEM DEFINITION

With the introduction of JIT compilation, two concepts that used to exist separately – code compilation and execution – were integrated into a single process. The on-demand fashion and the performance critical operation caused these concepts to be implemented in tight interaction. In the following we are going to discuss the operation of a typical JIT engine, the problems caused by its design and our threat model.

JIT Engine Operation. As an example for a JIT engine based on a conventional architecture, we refer to Nanojit, which is used by both Tracemonkey in Firefox and Tamarin in Adobe's Flash player. Figure 1 show the basic operation of a JIT engine.

Whenever code is being run by the *interpreter*, the *monitor* keeps track of how often certain code parts are being executed. Once the count for a code part reaches a certain threshold, the monitor tells the *recorder* to start recording a trace. The recorder keeps track of the execution, recording every instruction until a control flow branch takes it back to the part of the code that started the trace. Then, it passes the completed trace on to the *compiler*. The latter compiles the recorded trace from the interpreted language to native code and passes it to the *executor*. Now, whenever the monitor encounters a code part that matches the beginning of a compiled trace, it invokes the executor to run the native code instead of using the interpreter.

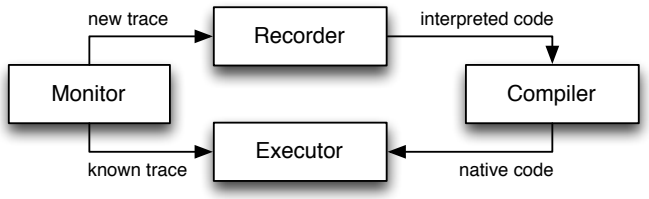

Fig. 1. Operation of a JIT Engine

Memory Access Rights. In a typical operating system, access to memory is restricted by permissions that are granted on a per-page basis. These permissions include read access (`R`), write access (`W`) and code execution (`X`). Memory pages can exhibit every combination of these access rights.

The issue with JIT engines is that the memory region the native code resides in has read, write and executable permissions set. This is a design issue that is dictated by the architectural layout of JIT engines: The compiler requires writes access to put the compiled code in the same region the executor needs execute permissions on to actually run the compiled code. This opens possibilities for attackers: if they manage to place their own native instructions in these memory regions, the instructions can be executed right away, given a vulnerability that allows to alter the control flow.

JIT Spraying and Threat Model. A JIT spraying attack consists of two stages. The goal of the first stage is to place "shellcode", i.e. code under the attacker's control, in an executable memory region. To this end, the attacker uses a series of instructions of the interpreted language that operate on large immediate operands, such as exclusive-or. When compiled to native code, these immediate operands are written unchanged into executable memory. Starting from the right offset, the corresponding bytes in memory decode to native code instructions. These instructions are used to form the shellcode. To actually get the JIT engine to compile the attacker's shellcode, the attacker places them into a loop so that they are executed often enough to pass the threshold for compilation. In the second stage, all that remains is for the attacker to leverage an exploit to redirect the JIT engine's control flow to the previously generated shellcode.

Based on the principles of a JIT spraying attack, our threat model is comprised of the following assumptions:

- The code of the interpreted language and thus the input to the JIT engine is under the attacker's control.
- The JIT engine's compiler features at least one vulnerability that allows the attacker to redirect the control flow to her benefit.

III. Approach

Our aim is to develop an architecture for Just-In-Time compilers of active content such as Javascript or Flash that is provably more secure against JIT spraying attacks than conventional architectures. In a nutshell, the problem with conventional architectures is that implementation flaws in the compiler can be leveraged to execute code that is injected in the course of compilation.

LOBOTOMY addresses this problem by splitting the JIT engine into two different processes: the compiler and the executor. While the compiler only has write access to memory pages used for compiled code, the executor only has execute permissions. Re-establishing this logical separation effectively prevents compiler bugs from being exploited to redirect control flow to code injected by a JIT spray.

This approach is based on the assumption that exploitable vulnerabilities are more likely found in the compiler than in the executor. The rationale behind this is that the compiler is more complex and thus bug-prone than the executor. We back this assumption by our evaluation that shows that a JIT engine runs significantly more different code parts in the compiler than in the executor (see Section VI-A).

IV. Splitting a JIT engine

In this section we present how we split the architecture of a typical JIT engine into two different processes. We are going to discuss how compiler and executor interact in LOBOTOMY from a conceptual point of view and during runtime.

A. Design

When splitting the JIT engine into two different processes, the tight interaction of the components needs to be re-established across process boundaries. We isolate the compiler from the executor by introducing separate execution contexts and use shared memory for synchronization and to exchange data.

Execution Contexts. Usually the JIT engine and all its logic components, that is monitor, recorder, compiler and executor run in one execution context. LOBOTOMY separates the execution context used for compilation from the one used for execution, with the context used for compilation being the main one that the execution context is being derived from.

Figure 2 shows the context switches involved in the execution of JITted code. First, the compiler clones itself to create the executor, both save their context and the executor waits on its semaphore (1-3). Then, interpreted code is executed (4) until a trace is ready for compilation and we switch back to the compiler (5, 6). After compilation, the compiler unblocks the executor (7) and the executor switches context again to execute the JITted code (8). When the JITted code is done, we again switch back to the executor (9, 10), which signals to the compiler (11). The compiler then switches back to the context where the JITted code stopped and resumes with interpreted code (12).

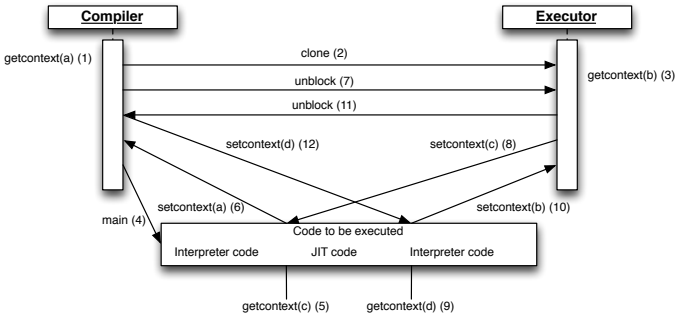

Fig. 2. Execution contexts involved in executing JIT code with LOBOTOMY.

Context switches are achieved by two system calls: `getcontext` will save the current context to a data structure that `setcontext` can resume from later on.

Shared Memory. To exchange data between the different execution contexts we use shared memory. Figure 3 depicts the two primary purposes shared memory serves:

- The modified compiler allocates JIT codepages from a shared memory file, mapped as RW. While the compiler maps the JIT codepages with RW permissions, the executor maps them as RX to prevent an attacker from loading more code into the page that is currently being executed.
- A shared memory object to allow easy communication between parent (monitor/compiler) and child (executor). It contains pointers to mutexes for synchronization, the state object, the trace to be executed, the side exit taken after the trace completes as well as other management information. It serves as the centerpiece of the communication between compiler and executor and is henceforth referred to as *communicator*.

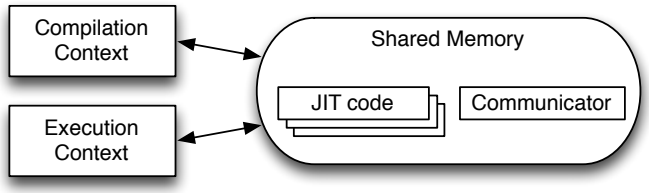

Fig. 3. LOBOTOMY uses shared memory to exchange data and synchronize the compilation and executor contexts.

B. Runtime Behavior

During monitoring and compilation time, the behavior of LOBOTOMY is the same as that of a conventional JIT engine. However, it deviates as soon as a compiled trace is about to be executed. In the following we first describe the normal process for trace execution before we present the differences to an execution sequence of LOBOTOMY.

Conventional Trace Execution. In order to execute a trace, first memory is allocated for the *execution state* and the *native stack*. The execution state is the object through which information such as stack- and frame pointers is passed to the trace. The native stack contains values used by the trace. The monitor then calls the trace as a function, with the state as parameter. When a trace terminates, it returns a so-called *side exit* which informs the monitor of the cause of the termination. In particular, it allows the monitor to keep a count of how many times a trace took a certain exit, such as a call to another piece of bytecode. If this count exceeds the above mentioned threshold, the monitor starts recording a new trace. The side exit is also used to keep the state of the interpreter and the native state synchronized.

While running a trace, potential exits are represented by so-called *guards*. A guard is an assumption made at the time the trace is compiled. As long as it holds, the control flow of the trace mirrors the control flow during the recording. When the assumption is broken, the control flow of the native execution is considered to have fallen off the trace, causing the compiled function to return. The monitor keeps track of these events by means of the side exits. If a new trace is recorded that represents the control flow beyond a guard, a jump to this trace is patched in instead of that guard.

Execution Sequence of LOBOTOMY. Figure 4 shows a detailed walk-through of an exemplary execution sequence. The lifetime of the object *compiler stack* is used to indicate that computations of both processes take place on the stack allocated for the compiler. The native stack, which the main function operates on, is only used when a process waits on a semaphore after a context switch. The basic execution sequence of the engine remains unchanged. However, we added the objects *compiler stack*, *compiler*, *executor stack* and *executor*. Correspondingly, we introduced calls (1) and (2) to generate the compiler stack and to start the compiler process. The process then follows the control flow until a trace is to be executed. At this point, we `clone` the compiler to create the executor process (10), which waits for the semaphore. Although it is not actually used, a stack for this process is allocated beforehand (9). After the compiler stores status information about the trace to be executed in the shared memory, a manual context switch, involving a `post` to a semaphore and a `setcontext` call (11) starts the executor. Now, the code compiled at runtime is executed (12). In our implementation, this implies a prior call to `mprotect` in order to make the corresponding memory executable. A more optimized implementation could map these memory segments into the executor context with code execution rights in the course of (11) or even (10). After trace completion, the communicator is used to store the trace information encapsulated in the TracerState object passed to the trace, as well as the exit taken to leave the trace. The executor then proceeds to unmarshal the data on the stack of the virtual machine back into the physical stack and updates several registers of the virtual machine. Finally, the executor saves its current context and triggers the context switch to allow the compiler to resume before pausing itself.

Built-in Functions. In interpreted languages, functions that are not implemented in the language itself, but rather in native code and are part of the runtime environment, are referred to as being *built-in*. JIT compiled code might include calls to such built-in functions, raising the question which context they belong to. As they typically make heavy use of the interpreter as well as the recorder's state, the compilation context is the

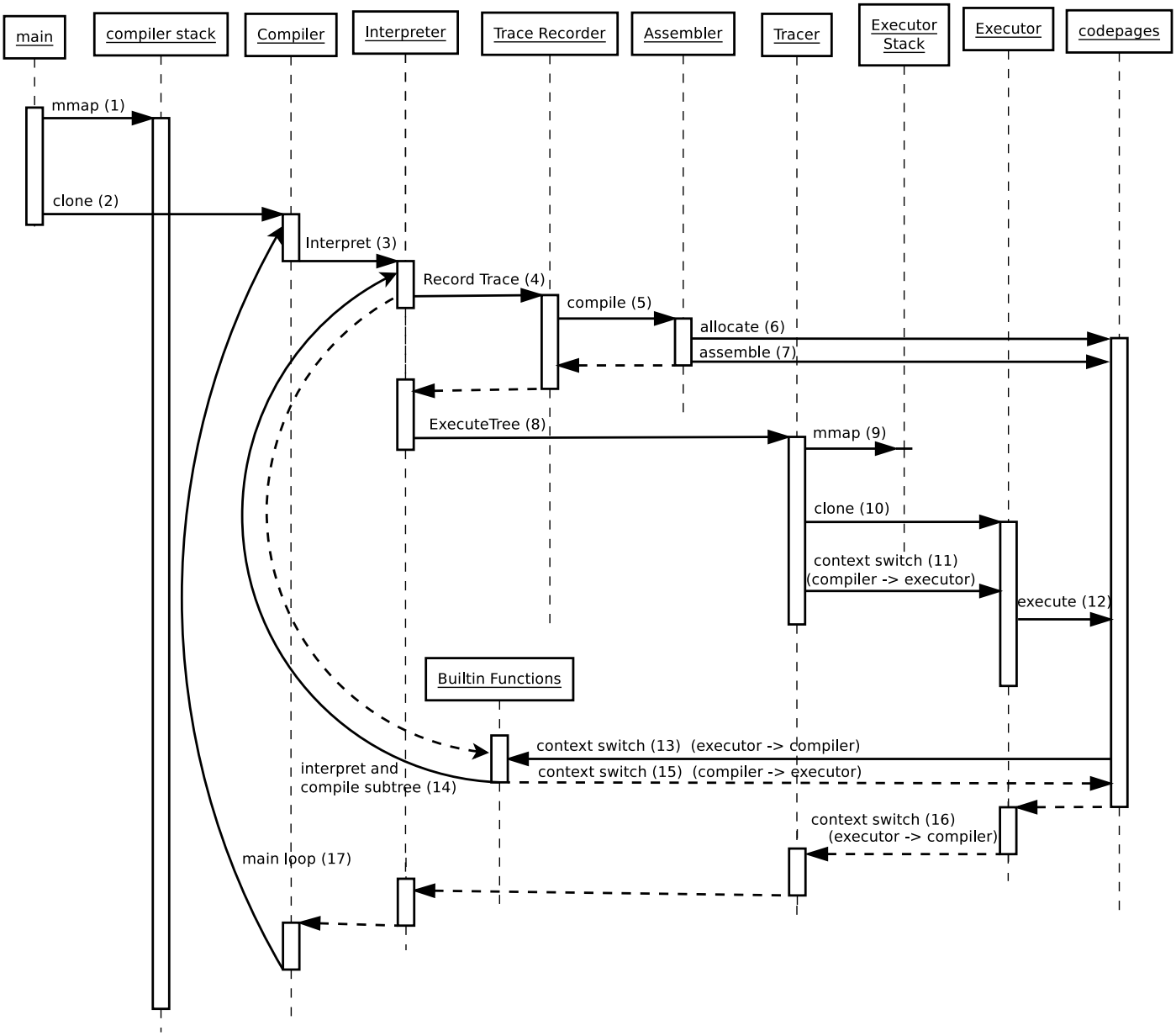

Fig. 4. Execution sequence of LOBOTOMY, including handling of built-in functions.

obvious choice. For this reason, we introduce a context switch from the executor to the compiler whenever a built-in function is invoked. After the switch, the compiler calls the built-in function requested by the executor. The built-in function may in turn call back to the interpreter itself (14), which might cause the recursive compilation of another trace. Once the recursive call returns, control is handed back to the executor running the JIT code (15). After completion of the JIT code, we switch back to the compiler context (16). From there on, LOBOTOMY functions in the same way like a conventional engine.

C. Multi-Processing Challenges

Several challenges have to be tackled in order to successfully implement our approach. First, it is necessary to make certain objects, such as the *runtime* object, which provides interfaces for memory allocation and similar tasks, available to two distinct processes. According to its documentation, Tracemonkey [8] is not designed for this. Thus, we have to ensure that these objects are accessed in a deterministic manner and that the two processes are indistinguishable from the runtime object's point of view. We also need to guarantee that all objects and variables that need to be shared by the two processes are shared. This obviously includes the memory region containing the compiled code and user data. It is equally important though that variables critical to the state of the JIT engine are shared as well. By default, these conditions and invariants are implemented as integer flags and reside on the stack.

Given these circumstances, a naive attempt to separate the executor from the compiler using normal processes as created by `fork` is likely to fail: The two components will eventually

loose sync because they do not share the same stack. In particular calls to a native C++ function from a trace often cause desynchronization. In the following a desynchronization likely leads to unexpected behavior and eventually to the violation of an assertion or even an outright crash.

While pointers to the codepages are fairly easy to localize and handle, it is more difficult to modify the allocation of the control flags. These flags are declared in header files and used all across the JIT engine. The engine expects these flags to behave and to interact in a certain manner, i.e. be located at certain offsets from each other. In fact, various sanity checks are based on these interactions. Splitting the engine into two different processes may break these invariants and cause the associated checks to fail.

The situation is further complicated by the fact that once the executor process is spawned, memory regions allocated by the parent are not automatically available in the child. In addition it is not always possible to reproduce an allocation of a chunk of memory at exactly the same address as in another process.

Since memory management is a critical component of Tracemonkey and Firefox in general, any modifications can have far-reaching and unforeseen consequences and should be kept at a minimum. Nonetheless, we require a custom memory manager underneath the memory management functions provided by Tracemonkey to manage the file-backed shared memory, which we need to allow implicit communication between the two processes after `clone`.

V. Implementation

Due to the complexity and high degree of optimization, we decided to keep changes to the JIT engine of Firefox as minimal and as local as possible. Thus, no interfaces between individual components were modified. However, some components had to be adapted in order to allow the engine to function when split into two parts. In particular, changes to the following elements are of consequence:

First, the Memory Manager had to be modified to permit the allocation of file-backed shared memory so that the two processes may share data and implicitly communicate through their objects. This also requires a solid API between the processes that allows one process to map memory that was mapped by the other process in exactly the same location. Second, a critical structure, containing much of the information that directs the control flow of the tracer was moved to the shared heap space. Finally, an explicit context switch had to be introduced at critical points in the code to ensure that as much native C++ code is executed by the compiler instead of the trace executor.

A. Memory Management

By default, Nanojit allocates all memory pages to store native code with RWX access rights. Due to this policy, no calls to `mprotect` have to be made after the initial allocation, beneficial to operational performance. However, this behavior is obviously no suitable basis for an architecture meant to be resistant to JIT spraying. Instead, the default access rights were changed to RW.

Since it is necessary to access the trace from across process boundaries, the trace and all data necessary for successful execution need to reside in shared memory. Memory management in Nanojit is handled by several allocators with different lifetimes and purposes, such as code, permanent data or temporary data. These allocators provide basic memory management functions such as allocate and free and work similar to the `malloc`-family of functions in the LibC. We modified these functions to use a custom heap management which is backed by a file in the shared memory space of the operating system. Our heap manager preallocates large sections of shared memory upon initialization and redistributes it in smaller blocks when required. While this might increase the initial footprint of Tracemonkey, the effect is expected to level down the longer the engine is running as Tracemonkey itself keeps large amounts of memory in reserve.

In some cases, either the executor or the compiler will allocate additional memory, which has to be made available to the other process. Thus, means for allocation synchronization are required. To this end, the `mmap` system call is used in conjunction with the `MAP_FIXED` flag, which allows the user to specify an exact base address for the chunk of memory to be allocated: When a process requests new memory, a chunk of memory is allocated from a shared memory file. The chunk's base address, its size and the offset in the memory file are written to the communicator object and a context switch is initiated. The other process can then use the information from the communicator object to allocate the exact same chunk and switch back to the original process.

Once compilation of a trace is completed, trace execution starts. If no executor exists yet, the process is spawned at this point. The memory region of the current trace as well as any new side exits that may have been generated by the compiler are then made executable for this process alone. Once these instructions have been made executable, the list used to access them is cleared. It is not necessary to keep a list of previously executed fragments, since in the memory space of the executor, the fragments stay executable during the executor's lifetime.

In addition to the changes to the allocators, some critical data structures such as `ThreadData`, containing status information of the tracer as well as data required by the Javascript virtual machine, were moved from the stack to the heap to allow them to be shared between the processes more easily. Still, some objects are required to remain on the stack. Fortunately data on the stack can be implicitly shared: since the two processes never run in parallel, they can inhibit the same stack area during most of their execution (see Section V-C).

B. Nested Traces and Deep Bail

A phenomenon that can occur during trace execution and that needs particular attention is nesting of traces, or rather, the abortion of nested traces. Traces are considered nested when the loops that caused them to be generated are nested. This means that the call graph of the trace consists of an outer and an inner graph, where the outer calls the inner graph [2]. This condition does not usually interfere with the modifications made for LOBOTOMY. However, trace abortion may cause complications in certain cases. As mentioned before, some

utility functions in the tracer are implemented directly in machine code in order to increase performance. These native functions can then call functions such as `js_Interpret`, which eventually cause a new trace to be recorded. Conceptionally, such traces are one trace with a call to an opaque, native block. In practice though, the trace may abort with a condition called a *Deep Bail*. In such a case, the control flow executes code associated with the compiler before returning from the trace, instead of directly jumping back to the exit point of the executor. The consequence is that the roles of executor and compiler are no longer as strictly disjoint as they would conceptionally be. Since the memory regions containing the native code are mapped into the executor with read- and execute permissions only, the compilation of this kind of trace could crash, were it not for well-placed context switches around the functions which cause such behavior. Additionally, this issue runs contrary to the design goal of isolating the code and memory used by the executor from areas used by the compiler, as far as possible.

To solve this problem, context switches from executor to the compiler were added to all functions that could cause nesting of traces and Deep Bails. The executor is reactivated after the function that caused the abortion of the trace returns. It is then merely responsible for updating the status object to signal a Deep Bail condition and to unmarshal the VM stack and the virtual machine's registers and then switches its context with the compiler.

We use context switching that is entirely based on `getcontext` and `setcontext`. The context saved by `getcontext` in one process is written to shared memory and used in `setcontext` by the other process. This requires that each process is able to operate on the stack of the other process, or more practically, that both processes share a stack. This is commonly thought of as troublesome, and the man-page of `clone` insists on the allocation of one stack per process.

C. Shared Stacks

Both the compiler and the executor must have separate stacks. However, `clone` allows the user to provide any suitably aligned and sized piece of memory for use as stack by the child entity, even if it is drawn from a shared memory file. To replace the physical machine's native stack with shared memory, it is necessary to pre-allocate two large blocks of data immediately after the application starts, so that no important information is stored on the native stack. After stack allocation, Tracemonkey must be made aware of the change of the stack's base address. This information is required to detect over-recursion and similar conditions, as well as to accurately calculate various offsets. Once the engine is aware of the change, the compiler proceeds to execute as usual. When a trace is about to be executed, a new process is cloned and the second pre-allocated memory chunk is assigned as its stack. This memory will, however, see no true use. The compiler, in the course of the context switch, saves its context, which includes the compiler's stack, and then jumps to a context on the native machine stack. The compiler's destination context is located on the native stack because this is where the first `getcontext` originated. After the compiler signals to the executor's semaphore, the executor switches its own context with the one just saved by the compiler, from where execution continues. This process works exactly the same way in the other direction and is triggered by either completion of the trace, a built-in function call or a request for memory allocation.

VI. Evaluation

In this section, we evaluate both the effectiveness and the performance of LOBOTOMY. We show:

- that the assumption LOBOTOMY's design is based on – less attack surface in the compiler than in the executor – is reasonable.
- the reduction of executable JIT memory regions during compilation compared to other approaches is significant.
- the performance of LOBOTOMY is competitive.

As a comparison baseline we use an unmodified version of the Javascript engine, both with JIT compilation en- and disabled. In addition to the baseline we also implemented the approach proposed by Chen et al. in Jitdefender [4]. Jitdefender tries to prevent JIT spraying by using only least-privilege permissions for the JIT codepages during compilation and execution. We implemented this approach by inserting an additional call to `mprotect` to add execute permission to the JIT memory regions before execution. Because of the shortcomings of the Jitdefender approach (see Section VIII), we also implemented an enhanced version that will re-protect the JIT memory regions when they are not being run: the access rights to side exits and previously compiled fragments are set and reset before and after trace execution. Throughout the evaluation we will refer to the basic Jitdefender approach as *NX* and the enhanced version as *re-protecting NX*.

A. Effectiveness

The effectiveness of LOBOTOMY's design is based on the assumption that the execution component of a JIT engine has a considerably smaller attack surface for memory corruption vulnerabilities than the compilation component. To provide proof of this assumption, we measured the ratio of code executed by the executor process in comparison to code executed in the compiler process. The code coverage evaluation is based on runtime data produced by the automated "JIT test" testsuite shipped with Tracemonkey, which consists of 1087 individual test cases.

For this purpose we use the program instrumentation tool PIN [11]. We modified the "proccount" PIN tool to keep record of invoked functions and the number of instructions executed per function. While this measurement does not provide instruction- or path coverage in the usual way, it does provide an approximation of the complexity of each process. This is sufficient to determine how much native code can potentially be run with execute rights to JIT code.

The results of the coverage evaluation are depicted in Table I. They clearly show that the separation between executor and compiler reduces the amount of code executed with execute permissions on any chunk of heap memory significantly. When running the set of regression tests, the number of functions executed by the executor is 5023 with

TABLE I. NUMBER OF FUNCTIONS AND INSTRUCTIONS (EXECUTED DURING THE FIRST RUN OF EACH FUNCTION) AFTER 1087 TEST CASES.

Process	Function Count	Instruction Count
Compiler	935k	251,106k
Executor	5k	150k

150,229 instructions. At the same time, the compiler executed 935,337 functions with over 250 million instructions.

While the call trace of the compiler shows all functions related to JIT compilation as well as the built-in functions executed during a JIT trace, the executor only shows functions that are necessary for the context switch, such as `setcontext` and `getcontext`, functions for memory allocation such as `mmap` and the functions initially called before built-in functionality is executed on trace. These invocations are related to tasks such as string concatenation, setting and getting of object properties or the execution of regular expressions. They remain on the trace of the executor since they have been replaced with stubs that trigger a context switch if they are called from the executor. Since these stubs are all generated by a small set of C++ macros, varying only in the number of parameters that are passed, this code is easy to audit. The rest of the executed code is either located in LibC or in the function `ExecuteTrace`. The latter encapsulates the start of the trace and the setting of execute rights to newly compiled code fragments and currently measures 63 lines of code.

In conclusion we provided evidence that LOBOTOMY reduces the attack surface of Tracemonkey for JIT spraying by at least two orders of magnitude. There are **no** code fragments that are mapped with RWX-permissions, because the virtual memory of the two processes is fully separated.

Comparison with NX. While LOBOTOMY does not allow for code to be mapped with RWX-permissions by design, the NX approach rather works on a best-effort basis. We demonstrate this by simulating a normal user session protected by the re-protecting NX variant. We use Selenium [9] in Firefox to automatically interact with common websites. Figure 5 shows the number of RWX pages in the course of the user session. The graph is characterized by a more or less linear increase, with some plateaus after new memory is allocated. Occasional, small drops are caused by garbage collection events.

As part of the test we visited several common websites, such as google.com, facebook.com, twitter.com, youtube.com and a local news portal. On each site, the mouse was hovered over each active element, including links, images with alt-texts and other components that were found. Finally we simulated interaction with maps.google.com, which caused the large increase of executable pages at the end of the test run.

The experiment both shows that even the re-protecting NX approach allows for a significant amount of memory regions to be mapped RWX. At the same time it also questions the effectiveness of threshold-based prevention approaches: while usually the amount of executable memory is in the region of hundreds of kilobytes, it can also spike even for benign sites.

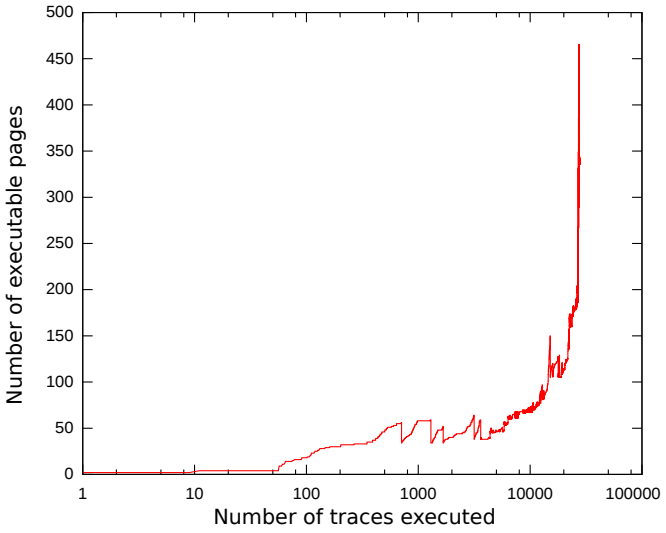

Fig. 5. Number of pages that remain write- and executable with re-protecting NX during simulation of a user session.

TABLE II. RUNTIME PERFORMANCE MEASUREMENTS.

Architecture	Runtime Overhead	95% Quantile
Baseline with JIT	1.00x	1.00x
Non-re-protecting NX	1.02x	1.02x
Baseline w/o JIT	2.60x	6.86x
LOBOTOMY	4.87x	1.27x
Re-protecting NX	9.04x	>10x

B. *Performance*

Table II shows the results of our runtime performance measurements. The implementation of the basic NX approach only introduces a negligible overhead, as it only adds a single system call before each trace execution. In contrast, its more secure counterpart that re-protects the executable memory regions, has a significantly higher overhead. This overhead is predominantly caused by the additional system calls to re-protect memory regions and the necessary bookkeeping. With LOBOTOMY, the sole fact that the engine consists of two processes that need to be synchronized causes an intrinsic performance overhead. This is not only because of the need for context switches, but also because any allocation of shared memory after `clone` needs to be performed twice.

For LOBOTOMY's performance evaluation, we not only show numbers on the mean runtime overhead, but also the 95% quantile. The reason are specific corner cases for particular test cases that result in a severely decreased performance. We thus regard the 95% quantile as being representative for the performance of the separated architecture. It shows LOBOTOMY is faster than the baseline without JIT and only adds a 0.27 overhead to the normal baseline.

The corner cases are not caused by the underlying design, but a limitation of our proof-of-concept implementation. To be specific, in our implementation Tracemonkey falls back to iterating over an internal list that is used for bookkeeping, instead of accessing the elements directly via a hashtable.

C. Proof-of-Concept Exploit

To showcase LOBOTOMY's effectiveness and its security gain in comparison to re-protecting NX, we chose CVE-2012-0464, a use-after-free vulnerability in the function `array_toString_sub` that is invoked by `array_join`. To exploit the vulnerability, the attacker has to overwrite the built-in function `toString` of an object inside an array. This function is then invoked whenever the array member with the overwritten method is processed by the `join`-function. This exploit causes a convoluted control flow in which the JIT compilation is triggered inside a recursive call from within another built-in function.

Since this could place an arbitrary constraint on an attacker's window of opportunity, we additionally simplified the vulnerability in order to cover a broader range of attack scenarios. To this end we inserted an additional indirect function call in the same basic block of code that also contains the original vulnerability.

LOBOTOMY renders both vulnerabilities, original and simplified ineffective. The pointer to the code snippet to be executed is transported using the shared communicator object. Even though exactly the same code that is executed by the executor process is called by the compiler in `array_toString_sub`, the call fails immediately due to insufficient access rights. This does not change even when we use the simpler exploit to trigger our vulnerability. We can therefore conclude that LOBOTOMY provides adequate protection from attacks that target previously JIT compiled code.

With the re-protecting NX approach the call to the JIT compiled fragment also fails for the original vulnerability. The reason for this failure is that the `Array.join` built-in function is called from the outer tree after the inner tree, in which case trace execution has already finished. Since the execute bit is unset after the traced tree finishes, the malicious call fails.

However, the simplified vulnerability, which generates a simple and straight forward trace that is compiled after the eighth iteration, causes the introduced jump to succeed. The reason behind is that the trace is still running and that the application had no time to unset the execute bit on the relevant codepages.

A single trace tree can consist of countless fragments that are linked together by the assembler. This happens when a previously "cold", that is, unused branch of the call tree becomes "hot" and a new fragment is compiled for this subtree. The fragment is then linked into the place previously occupied by a guard that causes the execution of the trace until its end. An attacker could therefore build a very large trace tree composed of individual fragments that are connected by branching conditionals. Execution of this function tree in a long-running loop would cause an increasing number of interlinked fragments to be added to the tree: As all fragments in a tree need to have the same access rights, execution of the root turns all children executable.

Hence, we conclude that the NX approach is not suitable to protect the JIT engine from attacks that target JIT compiled fragments.

VII. LIMITATIONS AND FUTURE WORK

LOBOTOMY has been built to be secure by design, thus limitations regarding its effectiveness are solely determined by the threat model. The threat model assumes that the executor is free of vulnerabilities. Thus, a possible scenario is a flaw in the execution part that allows the attacker to redirect control flow, although evaluation has shown that this is highly unlikely. In theory, an attacker could also use code reuse techniques after exploiting a bug in the compiler to invoke `mprotect` and mark the memory page containing the shellcode executable. However, if circumstances already allow for code reuse techniques to be used in the first place, a JIT spraying attack is not needed anyway.

The concept of separate address spaces per process is central to LOBOTOMY's design. It is thus not applicable to JIT engines that reside in the kernel of an OS, such as the Berkeley Packet Filters JIT engine of the Linux kernel.

In terms of performance, LOBOTOMY's biggest drawback is the inherent runtime overhead caused by switching between the compiler and the executor. Further careful optimization and analysis of the security requirements could help reduce the performance penalty by reducing the number of context switches. In particular, many built-in functions may prove to be harmless, such as functions that do not compile nested traces and that do not interact with data controlled by the user. Besides, the logic translating virtual machine instructions into the intermediate language could also be made aware of the separation between compiler and executor.

Apart from that our proof-of-concept implementation of LOBOTOMY currently fails on 18 of the test cases of Tracemonkey's regression test suite. We identified three categories of memory-management related errors that are caused by the complexity of splitting an already existing JIT engine:

- A mismatch between trace and statically compiled code. This happens if statically compiled code assumes dynamic objects from the trace to be located in a certain memory region, but the trace does not meet this requirement. Eliminating these errors would require a defined memory layout shared between statically and dynamically compiled code locations.
- Over-recursion. This occurs when a testcase runs out of stackspace. LOBOTOMY's current implementation uses a static stack size because this eases the necessary synchronization between compiler and executor.
- A fragment of dynamically compiled code that is being called in a trace is not executable. This happens when a compiled memory region is not tracked correctly and therefore not marked as executable at some point. As a result, the executor process will stop when the code fragment is encountered.

We would like to point out that these issues are not relevant to LOBOTOMY's effectiveness. Addressing them as well as the corner cases in our performance evaluation is a pure engineering task that requires an effort outside the scope of this paper. We thus leave it for future work.

VIII. Related Work

While disallowing RWX-accessible memory pages, as systems like SELinux do, does protect from JIT spraying, it will also break current JIT engines in general, as they require memory pages with write and execute rights.

Apart from that, many protection mechanisms against common heap spraying attacks fail when JIT spraying is involved. Egele et al. proposed a defense against heap spraying based on identifying shellcode in string buffers [7]. This mechanism cannot be applied here since no string buffers are used to store shellcode.

With NOZZLE [12], Ratanaworabhan et al. describe a general defense against heap spraying that tries to detect NOP sleds on the heap and attempts to disassemble objects on the heap. This technique's effectiveness is limited against JIT spraying attacks that use memory leaks to find their jump targets. Apart from that, JIT engines also tend to generate quite a large amount of data on the heap even under normal conditions. It may thus be possible to conduct a JIT spray while keeping the heap utilization beneath NOZZLE's threshold.

Bania [1] proposes a heuristic approach against JIT spraying, looking for a series of suspicious instructions in the disassembly of memory regions that store JITed code. While this heuristic seems reliable, we preferred a systematic approach to mitigate JIT spraying.

JITSEC [5] mitigates JIT spraying by adding callsite awareness to system calls implemented by the Linux kernel. Although this method can protect against current forms of JIT spraying, it is operating system dependent. Furthermore, future attacks might combine JIT spraying with code reuse techniques, which are not detected by this defense mechanism.

Rohlf et al. [13] discuss several ways of hardening JIT engines. While techniques such as constant blinding, which applies an XOR with a secret key to all constants, or NOP injection, which inserts random NOP instructions into the JIT compiled code could prove to be effective, they can often be disabled either by exploiting a memory leak in addition to the actual vulnerability or by spraying more malicious code.

Tao et al. [14] propose the insertion of code that is skipped if it is correctly aligned but triggers an interrupt if it is aligned incorrectly. The proposed defense mechanism also employs the randomization of register assignments and transforms the immediate operands of instructions. This defense strategy would prevent the usage of longer sections malicious, JIT code, but attackers could still use short snippets of malicious JIT code to gain control over the application.

Ping Chen et al. [4] attempt to prevent JIT spraying by selectively setting the execute permission for memory pages containing JIT code just before it is executed. In addition to the drawbacks of this approach shown in our evaluation, it ignores the fact that the execution of JIT code may recursively re-enter the interpreter function of a JIT compiler. In this case, memory would be executable during interpretation, which drastically reduces the gained security.

IX. Future Work and Conclusion

In this paper we presented Lobotomy, an approach to harden a JIT engine's design against JIT spraying attacks. In a nutshell it splits compilation and execution of JITed code into two strictly separated processes to reduce the attack surface. We implemented this approach as a working proof-of-concept in Tracemonkey, the JIT engine of Firefox. The evaluation shows that our modified design is not only successful in effectively reducing the attack surface, but also performs with a reasonable overhead.

Still, future work will focus on further optimizing our implementation performance-wise, with the ultimate goal of integrating it into a recent version of Firefox. At the time of writing, newer versions of Tracemonkey and its successors (Jaegermonkey and Ionmonkey) have already been released. Since they all build up on the architecture that was modified here, our concept can be adapted to these newer engines.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n. 257007 (SysSec).

References

[1] P. Bania. Jit spraying and mitigations. *arXiv preprint arXiv:1009.1038*, 2010.

[2] M. Bebenita, M. Chang, K. Manivannan, G. Wagner, M. Cintra, B. Mathiske, A. Gal, C. Wimmer, and M. Franz. Trace based compilation in interpreter-less execution environments. Technical report, ICS-TR-10-01, University of California, Irvine, 2010.

[3] D. Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *Blackhat, USA*, 2010.

[4] P. Chen, Y. Fang, B. Mao, and L. Xie. Jitdefender: A defense against jit spraying attacks. *IFIP Advances in Information and Communication Technology*, pages 142–153, 2011.

[5] W. De Groef, N. Nikiforakis, Y. Younan, and F. Piessens. Jitsec: Just-in-time security for code injection attacks. In *Proceedings of the Conference on Security and Privacy in Wireless and Mobile Networks*, 2010.

[6] DVLabs. EUSecWest Pwn2Own Contest. http://dvlabs.tippingpoint.com/blog/20120, 2012.

[7] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Berlin, Heidelberg, 2009.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the conference on Programming language design and implementation*, 2009.

[9] A. Holmes and M. Kellogg. Automating functional tests using selenium. In *Proceedings of the Agile Conference*, 2006.

[10] Mozilla. Firefox Bug 506693. https://bugzilla.mozilla.org/show_bug.cgi?id=506693, 2012.

[11] H. Patil, R. Cohn, M. Charney, R. Kapoor, and A. Sun. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proceedings of the Symposium on Microarchitecture*, 2004.

[12] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Microsoft Tech Report MSR-TR-2008-176*, 2008.

[13] C. Rohlf and Y. Ivnitskiy. The security challenges of client-side just-in-time engines. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[14] T. Wei, T. Wang, L. Duan, and J. Luo. Secure dynamic code generation against spraying. In *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.